VLSI SIGNAL PROCESSING

THROUGH BIT-SERIAL ARCHITECTURES AND SILICON COMPILATION

D. RENSHAW, B.Sc, M.Sc.

A Thesis submitted to the Faculty of Science of the

University of Edinburgh, for the degree of

Doctor of Philosophy.

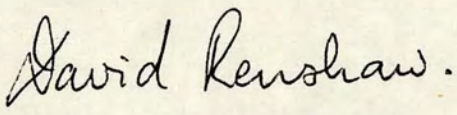
DEPARTMENT OF ELECTRICAL ENGINEERING

JULY 1984.

DECLARATION OF ORIGINALITY

I hereby declare that this thesis has been composed by myself. The work described has been developed by a group of researchers, including P. B. Denyer, N. W. Bergmann, A. F. Murray and S. G. Smith in addition to myself. I declare that I have made a major contribution to the work, being the only researcher with full-time involvement throughout the duration of the project, from start to finish. My main area of responsibility has been the development of the methodology, design environment specification and the design and verification of the cell library. I have, however, also contributed in all other areas including software, test methodology and system implementation.

Signed



David Renshaw

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. P. B. Denyer for his guidance, support and encouragement throughout the period of the research and study for this thesis. I would also like to thank the many people in both the departments of Electrical Engineering and Computer Science who have assisted me during this time. Special thanks are due to N. W. Bergmann S. G. Smith and J. G. Hughes.

I am grateful to the University of Edinburgh and to the Science and Engineering Research Council for providing facilities and financial support for this research. I would like to acknowledge the publishers Addison-Wesley and thank them for permission to reproduce items of text and illustration which will appear in "VLSI Signal Processing: A Bit-Serial Approach" by Peter B. Denyer and David Renshaw. I would also like to thank Dr. A. F. Murray for taking chip photographs, S. G. Smith for preparing Figures, Dr. A. R. Dinnis and J. Goodall for use of the Scanning Electron Microscope and Mrs C. A. Burns for secretarial assistance.

Finally I want to thank my wife Ann for her support and encouragement.

LIST OF CONTENTS: page no.

LIST OF CONTENTS (continued): page no.

LIST OF CONTENTS (continued): page no.

LIST OF CONTENTS (continued): page no.

LIST OF CONTENTS (continued): page no.

To my wife Ann.

CHAPTER 1

INTRODUCTION

This thesis is concerned with the problem of how to implement digital signal processing systems in silicon integrated circuit (IC) technology. Traditional methods for system design have typically relied on the following sequence of design actions. Initial system concepts are explored algorithmically by high level computer simulation. A prototype system is conceived in terms of existing standard components, typically families of TTL, etc. A prototype breadboard system is constructed. There then follows a period of testing debugging and redesign, even perhaps extending to modification of the system specification. Once the prototype breadboard is working it is then mapped directly into silicon ICs. This may be done using custom or semi-custom methods, depending on the size, complexity and bandwidth required of the system under design.

A number of problems attend the steps of this design sequence. Firstly breadboarding in standard parts imposes severe architectural and partitioning strictures. Secondly, there is no inherent structuring of design. Thirdly, there are few if any debugging aids. Fourthly, a direct mapping of the breadboarded prototype into silicon creates gross inefficiency in terms of silicon circuit design. In addition to these disadvantages, traditional methods of custom IC design are time consuming and costly.

By way of contrast, progress in IC fabrication technology has advanced by orders of magnitude, providing the technological capability of integrating hundreds of thousands of transistors onto a

single chip. This throws down a challenge to systems designers: that of designing systems requiring this complexity. In other words the two significant problems are how to design and what to design. The `what to design´ divides into two: mapping known and improved versions of known systems, and real innovation in the form of new system invention. The challenge of the `how to design´ is the discovery of methods for fast, efficient design generation. Added to this challenge of technology there are applications requirements for miniaturisation of systems for reasons of cost, weight and size, power dissipation and reliability; and there are economic pressures for fast cheap design.

One approach to the problem of how to design is to design programs and use existing computers for hardware. This correspond to a direct implementation of the high level simulation design. The advantage is that the design cycle is significantly reduced and that there are debugging aids and design can be inherently structured. There are also significant problems. Until very recently there were usually no fast enough, small enough, cheap enough computers. Movement in the direction of providing such computers has occurred. However, for the foreseeable future these will not be capable of addressing high bandwidth, computationally intense real-time applications. Further, they always possess inefficiency with respect to dedicated, or specific applications. In order to fulfil these requirements, custom design is still needed as it provides the only solution.

Typically the principal concerns of the system designer in addressing the custom implementation of a system are as follows. Firstly, he needs some method of design capture which can be used

to communicate all the necessary details of the system specification to the custom design process. Secondly he is interested in the time delay between submitting the design specification and receiving fabricated parts. Thirdly he is concerned about the cost. Fourthly he is concerned about testing the ICs returned to him. Here it should be remembered that control and observation of nodes on a breadboard of discrete parts is a totally different proposition to control and observation of nodes inside an IC. The details of silicon circuit design are not of interest to him except with respect to the testability of the circuit and with respect to power dissipation and the efficient use of silicon area.

Thus a requirement for rapid, cheap custom prototyping in silicon is identifiable. The objective of the research described here is to develop a design environment capable of providing this facility. The features it should offer will include the following. It must provide a simple but powerful language for system design capture. It must allow flexibility in system architecture and partitioning. It must provide a method for rapidly evaluating alternative systems prior to fabrication. Generation of mask geometries must be automated, but still reasonably efficient, and include a guarantee of performance. Testability must be in-built so that both the necessary test patterns are automatically generated and so that they give the fully required test coverage.

The contents of this thesis are arranged as follows. Chapter 2 gives a reviews signal processing systems, very large scale integration (VLSI) and computer aided design (CAD), in order to reveal the techniques adopted from these areas and so as to relate this research to that of other researchers. Chapter 3 describes

the design environment which has been developed. Chapters 4 and 5 give an outline of architectural design techniques, system synthesis and system mapping. These lead to efficient methods for system design capture. Chapter 6 gives a brief summary of designs completed and uses these to evaluate design capture. Chapter 7 describes the preliminary hardware fabrication and testing associated with the development of the design environment. Finally Chapter 8 is a short summary and conclusion.

There are four appendices. The first contains details of a version of the hardware description language. The second contains descriptions of an initial library of primitive constructs. The third contains an example case study of a system design. The fourth contains the authors publications related to this subject.

CHAPTER 2

SIGNAL PROCESSING SYSTEMS, VERY LARGE SCALE INTEGRATION AND COMPUTER AIDED DESIGN

2.1. Introduction

The research described in this thesis centres on establishing methods and tools for the implementation in custom, very large scale integrated (VLSI) circuits of digital signal processing systems using bit-serial architectures. As such, this research is interdisciplinary, - embracing areas of signal processing, integrated circuit engineering, system testability, VLSI design methodology and computer science. In this chapter a review of the relevant, directly related research areas is given. This is done so as to set in context the choice of direction, as set out in the subsequent chapters.

2.2. Signals & Signal Processing Systems

The terms signal processing system and signal have three established interpretations [1]. Firstly, systems can be defined to be networks of electrical components (resistors, capacitors, inductors and independent sources). Signals are then defined to be the voltages and currents on nodes of the network. This is the approach of classical circuit theory which is covered in many standard texts including, e.g. [2,3]. Secondly, instead of viewing the elements of a network as electrical components they can be interpreted as mathematical functions, or transforms (multipliers, integrators, differentiators etc). Signals are then viewed as functions of time, satisfying the equations determined by the system. Such an

approach is more general than that of circuit theory and has long been a topic of study in mathematics, physics, engineering and other fields. Again there is a well established body of knowledge in this domain as exemplified, e.g., in [4,5]. Thirdly, and more recently systems have come to be viewed as digital devices and signals as sequences of numbers. This approach can be used to approximately implement the previously known systems, through the theory and techniques of digital signal processing. Again there is well understood body of knowledge in this area with a large literature including the classical text [6]. Digital systems, however, extend beyond the limited structures of classical signal processing; these also span the area of computing and programmed instruction machines.

This thesis addresses aspects of digital signal processing.

2.3. Applications

Applications of signal processing cover an ever widening area, including communications, audio recording and broadcast, sonar and radar systems. More recently applications in geophysics, speech analysis, synthesis and recognition as well as image processing and recognition have become very active areas for research. A survey of applications can be found in [7]. The development of applications presupposes certain enabling bodies of knowledge. Firstly an experimental and theoretical understanding of the subject area is needed. Secondly, appropriate technology for system implementation is also needed. Both theoretical studies and the available technology impose direct constraints on implementations. Reciprocally, implementations can stimulate improvement of theoretical models and

provide directional pressure on the development of technology.

The research reported in this thesis is not application specific but is primarily concerned with developing a method for rapid system implementation. Such a method will be relevant to all the above mentioned application areas.

2.4. System Implementation

Implementations of digital signal processing systems can be either analog or digital. Such classification rests on the method of representation of time and signal within the system. Time can be represented as a continuous variable, or as a discrete variable. In sampled data systems time is a discrete variable. The signal (voltage or current etc.) can also be represented as a continuous or discrete varible. Signals which are quantised are discrete signals, referred to as digital signals. Analog systems operate on continuous signals which are either continuous or time-sampled. Digital systems operate on quantised, time-sampled data. In the past, because of the real-time bandwidth requirements of signal processing applications, systems have been dominated by analog implementations. There are, however, accuracy and matching problems associated with constructing systems from analog components. Digital elements have fundamental advantages with respect to accurate long term memory, elimination of the need for calibration-adjustment for drift and the flexibility of reprogrammability for new requirements. Initially, real-time digital signal processing was not feasible. With the development of micro chip technology and progress in device scaling and the corresponding increase in component performance, digital components can now be designed to

compete with analog devices both in respect of complexity, bandwidth and cost [8,9]. Further progress in these directions assures the superior performance of digital processing in ever increasing areas. Thus many analog systems are being and will increasingly be superseded by digital replacements.

Digital signal processing functions require computationally intensive algorithms. Many applications require high real-time bandwidths. It is these two demanding requirements which dominate implementation. For any given technology, the overriding requirement is to design efficient hardware capable of meeting bandwidth requirements.

The important area of real-time digital implementation of signal processing systems is addressed in this thesis.

Implementation is constrained "from above" by application area, the required bandwidth and algorithm development. "From below" implementation is constrained by the available technology. These externally imposed constraints dictate the limits on what systems can be implemented in hardware. Intrinsic to implementation are the problems of design, testability and cost.

Broad implementation alternatives are:

1. standard parts;
2. programmable and micro-programmable parts;
3. semi-custom design (fixed cell library and gate array);
4. full custom design.

Table 2.1 lists a number of criteria against which to assess any implementation method and Table 2.2 gives tentative evaluations

against the four methods. A precise cost function will depend on the market rates for each resource at the given time and their estimated rate of change over the period of the implementation.

SOME ASSESSMENT CRITERIA

Design time
Debug & redesign iterations
Management of complexity
Power dissipation
Size
Performance
Reliability
Testability
Cost of testing
Cost of production
Cost of design
Product security

Table 2.1

For digital signal processing, discrete component and full custom design have been the most widely used methods, partly because they have often been the only feasible methods. This situation is a result of the fact that real-time digital signal processing largely requires computationally intensive, fixed function, high speed processing, which cannot be achieved within the restrictions of low performance implementation techniques. The situation is gradually changing, as higher performance programmable digital signal processors are developed and as gate array technology improves. However there still remain a large number of systems which can only be implemented in custom designs.

COMPARISON BETWEEN IMPLEMENTATION METHODS

	TTL	μDSP	GATE ARRAY	CUSTOM	COMPILED
ARCHITECTURE CHOICE	flexible	fixed	limited	flexible	constrained
DESIGN TIME	high/med.	medium	low	very high	very low
DEBUG & REDESIGN	at design	at design	few	several	not necessary
COMPLEXITY MANAGEMENT	no aids	variety	some	few	comprehensive
POWER DISSIPATION	high	high	medium	low	low
SIZE	large	med./large	medium	smallest	small
BANDWIDTH	wide range	low/med.	medium	wide range	wide range
TESTABILITY	DEPENDS ON DESIGN				very good
TEST COST	high	high	medium	high	low
PRODUCT SECURITY	poor	poor	good	good	good

Table 2.2

The research described in this thesis addresses the area of custom design.

2.5. Architectures

Implementation is primarily concerned with design. This involves the organisation of components (sometimes thought of as resources) to create hardware which carries out a required function with the specified performance. In a formal sense design is derived from a specification. At the highest level the components or resources available can be classified as arithmetic, logic, storage, control and communications. The system architecture is defined to be the particular organisation and structure of these components in the system. Two contrasting examples of a system architecture are the von Neumann architecture and the systolic array architecture. In a similar way, complex components have an internal architecture comprising their circuit elements. Examples of component architectures can be found in multipliers, adders, RAMs, ROMs etc. A chip architecture may be a system architecture or a component architecture, the distinction is largely a question of complexity and self-containedness, and may be reflected in the degree of integration (MSI & LSI chips are component architectures, whereas LSI & VLSI chips may be system architectures). Examples of chip architectures can be found amongst the many micro-processors, controllers and other custom designed chips commercially available today.

Architectures can be classified in a variety of ways. Classifications are based on some feature of the resource organisation. Thus there are synchronous architectures and asynchronous (or

self-timed) architectures, a classification with respect to communication. Structuring of communication or processing gives rise to a classification into bit-serial or bit-parallel architectures. Classification with respect to control function distinguishes hard-wired and programmable architectures. As a further refinement there are SISD (single instruction single data stream), SIMD (single instruction multiple data), MISD (multiple instruction single data) and MIMD (multiple instruction multiple data) architectures [10]. Further, classification according to arithmetic processing is also possible giving fixed-point, floating-point, twos complement, canonic sign digit and distributed arithmetic, etc., architectures.

2.6. Architectures for Digital Signal Processing

The importance of architecture to the implementation of signal processing systems is evidenced by the many papers which emphasise it [11,12,13,14,15,16,17,18,19].

For any specific system, an evaluation of the alternative architectural possibilities can be made. Conversely, the classes of system suitable for implementation in a particular architecture can be investigated. Attempts to develop general purpose implementation methods must address classes of systems specific to one or more architectures.

The major concern of this research was to apply appropriate design automation techniques to the implementation of real-time signal processing systems. At the start of the project, the state of research in design automation recognised that short to medium term success was obtainable only through implementations which

targeted at constrained architectures [20,21]. Thus the correct choice of constrained architecture becomes crucial.

With respect to communication and processing, two fundamentally different architectural styles have been developed in signal processing systems. These are bit-serial architectures and bit-parallel architectures. In this section a brief comparison of the features of each is given. This is done to justify the choice of architecture.

The main physical features of bit-serial architectures are single wire communication and small area processing elements, (order one or order n, where n is the number of bits per word). Arithmetic, logic and communication all operate with a bit-wise organisation, which is extended in time for each full word. By contrast the main physical features of bit-parallel architectures are multi-wire communication buses and large area processing elements, (order n or order n squared). Arithmetic, logic and communication all operate with a word-wise organisation which is extended in space. The consequences of this are as follows.

1. Bit-serial architectures have order one or order n area compared with order n or order n squared area for bit-parallel architectures. This relates to communication area, to processing area and to control function area. Storage area is approximately equivalent for both.

2. As a result of the reduced number of circuit elements per unit function in bit-serial architectures, power dissipation might be expected to be reduced by a factor n as compared with bit-parallel architectures. However, there is usually additional

saving due to the absence of multiple bus drivers and organisational overheads.

3. Component operating speed, as measured by maximum clock rate for clocked schemes, is determined by the unit element maximum operating delay. In bit-serial parts this is usually that of a full adder. In any event it is independent of word-length. In bit-parallel parts the maximum operating delay is usually determined by carry propagation and depends on word-length (typically order n or order log n, at added area cost). This unit element maximum operating delay forms the upper bound on system operating speed. Consequently, much tighter pipelining is possible in bit-serial architectures than is the case in bit-parallel architectures. Another factor reinforcing this effect is the nature of communication delays in each architecture.

4. Cost per unit function is approximately inversely proportional to yield, which is approximately inversely proportional to an exponential function of area. Thus bit-serial components possess a considerable cost advantage per unit function.

5. Bit-serial function is best suited to fixed (dedicated or hard-wired) configuration, though general purpose architectures are being pioneered [22]. Bit-parallel architectures are dominated by the instruction based ALU or data path models. Programmability is inherently easy to build in, with low overheads. The flexibility which this allows gives a considerable advantage over the restrictions of a dedicated, non-reconfigurable architecture. Thus fixed function archi-

tectures are less common and in general bit-parallel architectures do have an advantage over bit-serial architectures in this respect.

6. With respect to overheads, bit-parallel architectures tend to require more overheads in the shape of communications busses, additional registers, and more complex control than is the case with bit-serial architectures.

7. In system design the importance of partitioning is recognised, as for example in [23,11]. Ease of partitioning is jointly a function of the amount of communication that crosses a partition boundary and the size of objects to be allocated to each partition. In respect of both of these, as also in terms of pin count, the bit-serial architectures have a marked advantage.

8. The cost of functional parallelism is much less in bit serial architectures than is the case in bit-parallel, as is exemplified in [24].

Other important criteria are modularity and extensibility. Both architectures can be organised to exhibit these. The cost is a function of the features outlined above and tends to be less for bit-serial architectures. Overall system performance will depend on the composite effects of the single feature cost functions outlined above. Simple metrics such as speed area product, and speed squared area product have been proposed [25] but these do not cover all aspects which need to be taken into consideration.

The advantages of bit-serial architectures have long been

recognised for dedicated applications [26] and many systems have been built using this architecture as evidenced by [27,28,29,30,31,32,33]. An architectural methodology for the design of bit-serial signal processors has been developed, as set out in [12]. This approach constituted a major influence on the research reported in this thesis. The wide range of architectural variants available within a bit-serial overall organisation are well set out in this text.

Recently, bit parallel architectures have become increasingly popular for low bandwidth applications. The trend in this area is to design general purpose programmable signal processors and to customise applications by programming them [34,35,36,37,38,39,40].

The research of this thesis has chosen to restrict its investigations to bit-serial, dedicated architectures principally for applications beyond the capabilities of the programmable, general purpose micro-processor.

2.7. IC Technology and System Implementation

System implementation is constrained by the hardware technology available. Over the past twenty years integrated circuit technology has progressed from single devices through small scale integration (SSI), medium scale integration (MSI), large scale integration (LSI), to very large scale integration (VLSI) and beyond. Definitions of levels of integration tend to vary from author to author. A rough characterisation in terms of transistor count might give SSI in the range ten to one hundred, MSI in the range one hundred to one thousand, LSI in the range one thousand to ten thousand and VLSI as being above ten thousand.

Integrated circuit engineering can be viewed as having two main components: design and manufacture. Originally design was simple (in terms of the number of devices, their logic and interconnection) It was dominated by circuit engineering and device physics. Manufacturing technology formed the research and development frontier. With progress in this technology the following changes took place. Firstly, the number of devices that could be placed on a chip increased by orders of magnitude. Secondly the manufacturing processes became well understood, replicable and characterised. The impact of these developments on design was that the technology rapidly became commensurate with the requirements for complex system implementation. The early design methods, appropriate to small scale circuit engineering, were not suited to dealing with systems comprising from tens or hundreds of thousands to millions of transistors. In this context, design has become a major research topic. There is a further fundamental reason for the importance of design research as expressed in [41], where Mead states:

> " ...After an evolution of six or more orders of magnitude in the most important metrics of the underlying technology, we are still using the same conceptualisation of computing that was common in the era of vacuum tubes and core memories. A quantitative improvement of many orders of magnitude makes a qualitative difference in the way one must conceptualise a field. What started as an evolution in cost reduction must, sooner or later, lead to a true revolution in our basic thinking."

Gradually, as design becomes more complex a division of labour becomes necessary. Thus circuit design engineering, system design engineering and fabrication engineering separate out and there is a requirement for efficient interfaces between these areas. In the early history of integrated circuit design, circuit engineering dominated chip design. For VLSI technology system design

engineering must dominate chip design, for reasons of economy, efficiency and testability.

A potential cost escalation and communication crises were foreseen in the area of chip design. To avoid these, and in conjunction with engineering education, an initiative in VLSI systems design was launched in the USA with considerable impact on subsequent trends in chip design. The objectives of this program are set out in [42] and its success can be gauged from the results, reviewed in [43]. The pedagogical program became accessible to a wide circle of engineers through the text [44].

The major evolutions resulting from this movement were as follows. Firstly processing was "factored" out from design, by setting up a silicon foundry service [45] and defining "interfaces" into this, so that engineers from small scale organisations as well as universities,- groups traditionally not directly involved with a processing or design house, - could gain access to fabrication. Secondly standards for data format were proposed [44,46,47] and used; these are the simplified design rules and caltech intermediate form (CIF). Thirdly computer programming and the methodology of software design started to have a direct impact on chip design. Further discussion of this last influence is presented in the next section.

These developments opened up the area of VLSI systems on silicon. This thesis attempts to contribute to this area.

Historically the developments leading to VLSI were associated with MOS technology. Further, the implementation technology made available for this research was nMOS technology. Thus an

evaluation and choice of technology are not subjects addressed here but can be found adequately treated elsewhere, e.g. [48,49]. There are, however, issues of "technology binding" which are nonetheless of major importance. These concern the re-implementation of designed systems in a new technology as this becomes available and the cost and time involved in doing this. Any research into design must take account of this.

The objective of the research reported in this thesis is to develop means for designing complex signal processing systems in silicon so as to enable system engineers to have easy access to integrated circuit technology, without the expense and delay of custom design. This program of research is based on the assumption that there is and will continue to be a trend towards rapid, cheap silicon fabrication for small volume proto-typing and experimental system development, as has been seen in the USA.

It is now appropriate to list the known VLSI problem areas. Firstly there are the technology and scaling problems [50]. Secondly there are the timing, clocking and communications problems [51,52]. Thirdly there are power distribution and dissipation problems associated with scaling, increased speed and increased circuit density [50,53]. Finally, there are the design associated problems [41,54,55,56,57]. These include: algorithm development, complexity management, specification, partitioning, technology independence and technology binding, physical design generation (including floor planning, placement and routing) simulation, verification, testing, documentation and finally design time and cost

The first three areas require combined technology and design developed solutions, and are not areas which this research addresses. The remaining chapters propose and evaluate one possible solution to the design associated problems.

2.8. VLSI Design Methodology and Computer Aided Design

The ideas developed in design methodology and computer aided design have been applied, in order to solve the design problems. In this section a brief review of the key ideas is given.

Until the mid 1970s, CAD for IC design consisted of dissociated specific programs, mainly graphics editors, circuit simulators and data post-processing programs, including some layout rule checking [58]. As the level of integration increased, many problems in using these aids started to become evident, until it became clear that the currently available aids could not handle the complexity of the circuits to be designed. Also, a cost escalation crisis for custom design precipitated pressure to develop new ways of designing integrated circuits. A major problem of traditional custom design was the cost required for the necessary number of man years per project. This comprised a large initial value with perhaps the same again for debugging and redesign before a successful product could be released. It was imperative that new techniques and tools be invented. This situation inspired research into design methodologies and lead to structured, hierarchical design and a drive towards constructing methodology based CAD tools. Only in this way could complexity be handled, design cycles shortened and costs cut. Another key concept was that of bound structural, behavioural and physical descriptions [59,60]. As a

result, anything other than an integrated, hierarchical design environment is now seen as both inadequate and unsatisfactory. This was the first significant trend to influence the present decade of CAD - namely a movement towards structured, hierarchical and methodology-based, integrated design environments.

The second trend occurred simultaneously but if anything has even more importance. This trend originated from the area of programming languages and artificial intelligence. Circuits are composed of collections of comparatively small cells. These cells can be viewed as families of related variants. Instead of generating chip designs using graphics aids by placing fixed instances of cells, a structure of parameterised cells can be generated by computer program, through procedural definition. Implementation of this idea leads to creation of embedded languages for IC design. Early examples of these can be found in [61,62]. Two main types of embedded language have been developed: translation based languages and data structure based languages (cp. chapter 3 of [63]). In the former, graphic primitives are output as they are encountered. In the latter, a data structure is built up for the entire design before it is output.

The concept that chip design can be done by writing computer programs which generate the mask geometries has had "revolutionary" importance. This can best be understood by looking at the process of composing a chip from small previously defined cells, usually called leaf cells. In this discussion a comparison is made between on the one hand graphics based cell design & assembly and on the other hand design & assembly based on programming languages. Classes of complex structures, especially arrays for example, can

be efficiently described in terms of loops and conditionals. Further, if the definitions of the pre-defined cells are not fixed, but are parameterised, then evaluation of parameters can occur within such loop and conditional expressions. By way of contrast, the assembly of fixed cells by means of graphics editing requires a large collection of cells and a time consuming process of assembly. Further it results in only a single structure. Additionally, the size of the cells to be assembled has an upper bound imposed by the graphics system screens and operator limitations. By contrast, a greater range of cell size can be generated by parameterised procedural definition. More significantly, if cell definition requires instantiation of all its internals then it is fixed and, on assembly, no alteration is possible. Potentially much more efficient layouts can be generated if there is the flexibility to absorb global assembly constraints into cell definitions. With parameterisation of definitions there is the possibility to do this and the way is opened for computerised search for optimised solutions, if suitable algorithms can be found. This approach, however, is at present in its infancy. Developments along these lines lead through silicon assembly (which is usually based on translator embedded languages) to silicon compilation [63-80]. Some of these have since reached commercial "maturity" [78,81,82,83]. The latter, if based on a data structure embedded language, does have the potential to search for optimisation. The success of this approach to chip design was evidenced by the early designs generated by this method and their successful fabrication [84,85,20]. Finally, this development has opened the way for the development of the so called "intelligent, knowledge-based systems" for IC design [86].

The early silicon compilers have all been architecture specific. Their success was possible because of this. The first architecture to be investigated was the data path architecture [63,21]. Subsequently the application of the ideas of silicon compilation have been applied to gate arrays [78] and to a variety of signal processing architectures [20,87,88]. Concurrently with these developments the research reported in this thesis was under development. Progress has been reported elsewhere in [89,90,91,92,93,94].

Another important strand of research, suitable for incorporation into the framework of silicon compilation applied to system design, can be found in the high level algorithmic research reported in [56,95,96,97,98,99,100].

The main short-comings of the early silicon compilers were that they addressed, almost exclusively only the physical design problem. In signal processing, performance is a dominant constraint. To date the conventional silicon compiler approach to performance has been to "post-design" analyse it, or ignore the problem. The problems of binding to behavioural description, verification, performance and testability are crucially important areas which were not adequately addressed. Further inadequacy can be seen in the way technology binding was achieved and the problems of translation to a new technology.

The research reported in the remainder of this thesis has the objective of addressing with equal emphasis structural, physical, behavioural, performance and testability requirements. The tools

and techniques for achieving this were assessed as being attainable in the three year time scale. The problem of technology binding remained only partially solved, mainly because the prerequisites for a satisfactory solution were not available at the start of the project. Progress in this direction has, however been made and is reported in [101,102].

2.9. Summary

A brief review of the background research areas of relevance to this thesis has been given. The objectives of the research have also been identified and set in context. These may be summarised as the development of methods and tools which permit the rapid, "first-time right" design, the evaluation and the implementation of complex digital processing systems in IC technology. The approach adopted incorporates the use of program language definition for specification and a method (the system mapping method) for transforming a class of general algorithms into this language. It utilises the techniques of embedded languages and parameterised procedural generation of physical design. It features bound functional, behavioural and structural descriptions with an economic verification strategy, based on simulation and data base "proving". Performance is addressed from the outset through the use of custom, parameterised macro cell engineering, through the definition of communication standards and through the bit-serial, pipelined architecture chosen. Spectacularly economical system testability is achievable as a result of the architecture chosen and through the use of the system mapping method. For reasons of availability and widespread use, the implementation technology used is five micron nMOS polysilicon gate technology. The arithmetic format

chosen is fixed point two's complement and communication follows standard, two phase, non-overlapping clocked design. The realisation of these objectives and an evaluation of the results is presented in the remainder of this thesis

It should be noted that there are two external assumptions upon which the success of this project will be based. These are, firstly, that fast turn around silicon processing is available for testing and proto-typing and, secondly, that the cost of such a facility will be comparatively cheap, or at least competitive with that of the alternatives.

References

1. A. Papoulis, Circuits and Systems: A modern Approach, Holt-Saunders (1980).

2. P. R. Adby, Applied Circuit Theory, Wiley (1980).

3. J. Millman, Microelectronics: Digital and Analog Circuits and Systems, McGraw-Hill (1979).

4. T. Kailath, Linear Systems, Prentice Hall (1980).

5. M. Schwartz and L. Shaw, Signal Processing, McGraw-Hill (1975).

6. L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice Hall (1975).

7. A. V. Oppenheim (Ed.), Applications of Digital Signal Processing, Prentice Hall (1978).

8. C. M. Rader, "On Digital Filtering," IEEE Transactions on Audio and Electronics Vol. AU-16(3) pp. 303-313 (September 1968).

9. H. J. Whitehouse and K. Bromley, "Can Analog Signal Processing Survive the VHSIC Challenge?," Microwave Systems News, pp. 91-98 (April 1981).

10. M. J. Flynn, "Very High Speed Computing Systems," Proc IEEE Vol. 54 pp. 1901-1909 (December 1966).

11. E. E. Swartzlander Jr., "VLSI Architecture," pp. 178-221 in Very Large Scale Integration (VLSI) Fundamentals and Applications, ed. D. F. Barbe,Springer Verlag (1980).

12. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," pp. 131-140 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

13. J. Allen, "VLSI Architectures for Signal Processing," pp. 242-254 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

14. B. A. Bowen and W. R. Brown, VLSI Systems Design for Signal Processing, Prentice Hall (1982).

15. P. B. Denyer, "An Introduction to Bit-Serial Architectures for VLSI Signal Processing," pp. 225-241 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

16. P. R. Cappello and K. Steiglitz, "Completely-Pipelined Architectures for Digital Signal Processing," IEEE Transactions of Acoustics, Speech and Signal Processing Vol. ASSP-31(4) pp. 1016-1023 (August 1983).

17. P. C. Treleaven, "Decentralised Computer Architectures for VLSI," pp. 348-380 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

18. D. J. Kinniment, "VLSI and Machine Architecture," pp. 24-33 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

19. F. Anceau, "VLSI-Processor Architecture and Design," pp. 138-148 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

20. J. M. Siskind, J. R. Southard, and K. W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," pp. 28-39 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1981).

21. H. E. Schrobe, "The Data Path Generator," pp. 175-181 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1981).

22. W. P. Burleson, "A Programable Bit-Serial Processing Chip," M. Sc. Thesis, M.I.T. (1983).

23. M. Stefik, D. G. Borrow, A. Bell, H. Brown, L. Conway, and C. Tong, "The Partitioning Concerns in Digital System Design," pp. 43-52 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1981).

24. P. P. Reusens, R. W. Linderman, and W. H. Ku, "CUSP: A Custom VLSI Processor for Digital Signal Processing Based on the Fast Fourier Transform," Proceedings, IEEE International Conference on Computer Design, (1983).

25. J. E. Savage, "Planar Circuit Complexity and the Preformance of VLSI Algorithms," pp. 61-68 in VLSI Systems and Computations, ed. H. T. Kung, B. Sproull, G. Steele,Springer Verlag (1981).

26. L. B. Jackson, J. F. Kaiser, and H. S. McDonald, "An Approach to the Implementation of Digital Filters," IEEE Transactions on Audio and Electroacoustics Vol. AU-16(3) pp. 413-421 (September 1968).

27. N. R. Powell and J. M. Irwin, "Signal Processing with Bit-Serial Word-Parallel Architectures," SPIE Vol. 154 Real Time Signal Processing pp. 98-104 (1978).

28. N. R. Powell, "Functional Parallelism in VLSI Systems and Computations," pp. 41-49 in VLSI Systems and Computations, ed. H. T. Kung, B. Sproull, G. Steele,Springer Verlag (1981).

29. A. P. Reeves and J. D. Bruner, "Efficient Function Implementation for Bit-Serial, Parallel Processors," IEEE Transactions on Computers Vol. C-29(9) pp. 841-844 (September 1980).

30. M. R. Buric and C. A. Mead, "Bit-Serial Inner Product Processors in VLSI," Proceedings, Caltech Conference on VLSI, (1981).

31. R. W. Linderman, P. P. Reusens, P. M. Chau, and W. H. Ku, "Digital Signal Processing Capabilities of CUSP, a High Performance Bit-Serial VLSI Processor," Proc. IEEE ICASSP'84, pp. 16.1.1 - 16.1.4 (San Diego, March 1984).

32. Norihiko Ohwada, Tadakatsu Kimura, and Masanobu Doken, "LSI7s for Digital Signal Processing," IEEE Transactions on Electron Devices Vol. ED-26(4) pp. 292-298 (April 1979).

33. D. J. Myers and P. A. Ivey, "STAR -- A VLSI Architecture for Signal Processing," pp. 179-183 in Proceedings, Conference on

Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1984).

34. Akira Sawai, "Programmable LSI Digital Signal Processor Development," pp. 29-40 in VLSI Systems and Computations, ed. H. T. Kung, B. Sproull, G. Steele,Springer Verlag (1981).

35. B. Fette, D. Harrison, D. Olson, and S. P. Allen, "A Family of Special Purpose Micro-Programable Digital Signal Processor IC's in an LPC Vocoder System," IEEE Journal of Solid State Circuits Vol. SC-18(1) pp. 25-33 (February 1983).

36. S. P. Pope, B. Solberg, and R. W. Brodersen, A Single-Chip LPC Vocoder, Department of Electrical Engineering and Computer Science, University of California, Berkeley ().

37. TMS32010 Data Manual, Texas Instruments (1982).

38. Intel 2920 Signal Processor, Intel (1983).

39. "AMD2900 Family CPU, Sequencers, Peripoherals, Interface," in Advanced Micro Devices Data Book, Advanced Micro Devices (1982).

40. K. Okada, T. Ehara, H. Suzuki, K. Yanagida, K. Saito, and N. Ichiura, "A Digital Signal Processor Module Architecture and its Implementation Using VLSI," Proc. IEEE ICASSP'84, pp. 44.5.1 - 44.5.4 (San Diego, March 1984).

41. C. Mead, "VLSI and the Foundations of Computing," in Information Processing 83, ed. R. E. A. Mason,Elsevier Science Publishers (September 1983).

42. L.Conway, A. Bell, and M. E. Newell, "MPC79: The Large-Scale Demonstration of a New Way to Create Systems in Silicon," Lambda Vol. 1(2) pp. 10-19 (2-nd Quarter 1980).

43. G. Lewicki, D. Cohen, P. Losleben, and D. Trotter, "MOSIS Present and Future," pp. 124-128 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1984).

44. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley (1980).

45. D. J. Fairbairn, "The Silicon Foundry: Concepts and Reality," Lambda Vol. 2(1) pp. 16-26 (First Quarter 1981).

46. R. F. Lyon, "Simplified Design Rules for VLSI Layouts," Lambda Vol. 2(1) pp. 54-59 (1-st Quarter 1980).

47. C. H. Sequin, "Generalised IC Layout Rules and Layout Representations," pp. 13-23 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

48. A. Barna, VHSIC: Technologies and Tradeoffs, Wiley (1981).

49. A. B. Glaser and G. E. Subak-Sharpe, Integrated Circuit Engineering, Addison Wesley (1979).

50. J. L. Prince, "VLSI Device Fundamentals," pp. 4-41 in Very Large Scale Integration (VLSI) Fundamentals and Applications, ed. D. F. Barbe,Springer Verlag (1980).

51. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley (1980). Chapter 7 on System Timing by C. Seitz

52. J. Alves Marques and A. Cunha, "Clocking of VLSI Circuits," pp. 165-178 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

53. W. S. Song and L. A. Glasser, "Power Distribution Techniques for VLSI Circuits," pp. 45-52 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1984).

54. C. Seitz, "Ensemble Architectures for VLSI: A Survey and Taxonomy," pp. 130-135 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1981).

55. M. Rem, "The VLSI Challenge: Complexity Bridling," pp. 65-74 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

56. H. T. Kung, "Let's Design Algorithms for VLSI Systems," Department of Computer Science Report, Carnegie Mellon University (1979).

57. P. Losleben, "Computer Aided Design for VLSI," pp. 89-127 in Very Large Scale Integration (VLSI) Fundamentals and Applications, ed. D. F. Barbe,Springer Verlag (1980).

58. A. R. Newton, "Computer Aided Design of VLSI Circuits," Proceedings, IEEE Vol. 69(10) pp. 1189-1199 (October 1981).

59. I. Buchanan, "Modelling and Verification in Structured Integrated Circuit Design," PhD Thesis, University of Edinburgh, Computer Science Department (July 1980).

60. C. A. Mead, "Structural and Behavioural Composition of VLSI," pp. 3-8 in Proceedings of the IFIP International Conference on VLSI, ed. F. Anceau, E. J. Aas,North Holland (1983).

61. B. Locanthi, "LAP: A Simula Package for IC Layout," Caltech SSP Report, California Institue of Technology (1978).

62. R. F. Ayers, "IC Design under ICL, Version 1," Caltech SSP Report, California Institue of Technology (1978).

63. D. L. Johannsen, "Silicon Compilation," PhD Thesis, California Institue of Technology (1981).

64. C. R. Rupp, "Components of a Silicon Compiler," pp. 227-236 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

65. R. F. Ayers, VLSI Silicon Compilation and the art of Automatic Chip Design, Prentice Hall (1983).

66. F. Anceau and J. P. Schoellkopf, "CAPRI: A Silicon Compiler for VLSI Circuits Specified by Algorithms," pp. 149-154 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

67. F. Anceau, "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms," pp. 15-31 in 3-rd Caltech Conference on VLSI, ed. R. Bryant,Computer Science Press (1983).

68. A. A. Szepieniec, "SAGA: An Experimental Silicon Compiler," 19-th Design Automation Conference, pp. 365-370 (1982).

69. G. Brebner and D. Buchanan, "On Compiling Structural Descriptions to Floorplans," Proceedings, IEEE International Conference on Computer Aided Design, (1982).

70. M. R. Buric, C. Christensen, and T. G. Matheson, "The Plex Project: VLSI Layouts of Microcomputers Generated by a Computer Program," IEEE International Conference on Computer Aided Design, (September 1983).

71. T. G. Matheson, M. R. Buric, and C. Christensen, "Embedding Electrical and Geometric Constraints in Hierarchical Circuit Layout Generators," IEEE International Conference on Computer Aided Design, (September 1983).

72. A. M. Peskin, "Towards a Silicon Compiler," Proceedings, Custom Integrated Circuits Conference, (1982).

73. T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: Prototype System," Proceedings, 20-th Design Automation Conference, (1983).

74. D. E. Thomas, C. Y. Hitchcock, T. J. Kowalski, J. V. Rajan, and R. A. Walker, "Automatic Data Path Synthesis," IEEE Computer, pp. 59-69 (December 1983).

75. J. Werner, "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?," VLSI Design Vol. III(5)(September/October 1982).

76. J. Werner, Silicon Compiler Part 1: The Front End"" "Progress Towards the "Ideal" Silicon Compiler Part 1: The Front End," VLSI Vol. IV(5) pp. 38-41 (September 1983).

77. J. Werner, Silicon Compiler Part 2: Automatic IC Layout"" "Progress Towards the "Ideal" Silicon Compiler Part 2: Automatic IC Layout," VLSI Vol. IV(6) pp. 78-81 (October 1983).

78. J. P. Gray, I. Buchanan, and P. S. Robertson, "Designing Gate Arrays Using a Silicon Compiler," Proceedings, 19-th Design Automation Conference, pp. 377-383 (1982).

79. A. R. Deas, "The UNIT Silicon Compiler," MSc Thesis, University of Edinburgh, Computer Science Department (September 1983).

80. A. R. Deas, "Silicon Compilation: A VLSI Complexity Management Strategy," Proceedings, Euromicro '84 Conference, (August 1984, Copenhagen).

81. P. Wallich, "On the Horizon: Fast Chips Quickly," IEEE Spectrum, pp. 28-34 (March 1984).

82. S. C. Johnson, "VLSI Cicuit Design Reaches the level of Architectural Description," Electronics, pp. 121-128 (3-rd May 1984).

83. J. R. Fox, "The MacPitts Silicon Compiler: A view from the Telecommunications Industry ," VLSI Design Vol. IV(3) pp. 30-37 (May/June 1983).

84. R. A. Rivest, "A Description of a Single Chip Implementation of the RSA Cipher," Lambda Vol. 1(3) pp. 14-18 (4-th Quarter 1980).

85. J. Batali, E. Goodhue, C. Hanson, H. E. Schrobe, R. M. Stallman, and G. J. Sussman, "The SCHEME-81 Architecture -- System and Chip," pp. 69-77 in Proceedings, Conference on Advanced Research in VLSI, ed. P. Penfield Jr.,Artech House Inc. (1981).

86. H. E. Schrobe, "AI Meets CAD," pp. 387-399 in Proceedings of the IFIP International Conference on VLSI, ed. F. Anceau, E. J. Aas,North Holland (1983).

87. S. P. Pope and R. W. Brodersen, "Macrocell Design for Concurrent Signal Processing," pp. 395-412 in 3-rd Caltech Conference on VLSI, ed. R. Bryant,Computer Science Press (1983).

88. P. A. Ruetz, S. P. Pope, and R. W. Brodersen, Computer Generation of Digital Filter Banks, University of California, Berkley (March 1984).

89. P. B. Denyer, D. Renshaw, and N. Bergmann, "A Silicon Compiler for VLSI Signal Processors," Proc. ESSCIRC'82, pp. 215 - 218 (Brussel, September 1982).

90. N. W. Bergmann, "A Case Study of the FIRST Silicon Compiler," pp. 413-430 in 3-rd Caltech Conference on VLSI, ed. R. Bryant,Computer Science Press (1983).

91. P. B. Denyer and D. Renshaw, "Case Studies in VLSI Signal Processing Using a Silicon Compiler," Proc. IEEE ICASSP'83, pp. 939 - 942 (Boston, April 1983).

92. A. F. Murray, P. B. Denyer, and D. Renshaw, "Self-Testing in Bit-Serial Parts: High Coverage at Low Cost," Proc. IEEE International Test Conf., pp. 260 - 268 (Philadelphia, October 1983).

93. G. H. Allen, P. B. Denyer, and D. Renshaw, "A Bit-Serial Linear Array DFT," Proc. IEEE ICASSP'84, pp. 41A.1.1 - 41A.1.4 (San Diego, March 1984).

94. L. E. Turner, P. B. Denyer, and D. Renshaw, "A Bit Serial LDI Recursive Digital Filter," Proc. IEEE ICASSP'84, pp. 41A.3.1 - 41A.3.4 (San Diego, March 1984).

95. S. Y. Kung, "VLSI Array Processor for Signal Processing," Conference on Advanced Research in Integrated Circuits, MIT (1980).

96. S. Y. Kung, "Wave Front Array Processor: Language, Architecture and Applications," IEEE Transactions on Computers Vol. C-31(11) pp. 1054-1065 (November 1982).

97. S. Y. Kung, "From Transversal Filter to VLSI Wavefront array," pp. 247-261 in Proceedings of the IFIP International Conference on VLSI, ed. F. Anceau, E. J. Aas,North Holland (1983).

98. D. Cohen, "Mathematical Approach to Computational Networks," Technical Report, University of Southern California, Information Sciences Institute (November 1978).

99. L. Johnsson and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," pp. 213-68 in VLSI Systems and Computations, ed. H. T. Kung,

B. Sproull, G. Steele,Springer Verlag (1981).

100. L. Johnsson, U. Weiser, D. Cohen, and A. L. Davis, "Towards a Formal Treatment of VLSI Arrays," Technical Report, California Institute of Technology, Computer Science Department (January 1981).

101. N. W. Bergmann, "Idiomatic Integrated Circuit Design," PhD Thesis, University of Edinburgh, Computer Science Department (1984).

102. B. Ackland and N. Weste, "An Automatic Assembly Tool for Virtual Grid Symbolic Layout," pp. 457-466 in Proceedings of the IFIP International Conference on VLSI, ed. F. Anceau, E. J. Aas,North Holland (1983).

CHAPTER 3

A DESIGN ENVIRONMENT FOR GENERATING CUSTOM ICs TO IMPLEMENT DIGITAL SIGNAL PROCESSING SYSTEMS IN BIT-SERIAL ARCHITECTURES

3.1. Introduction

This chapter describes a design environment for implementing digital signal processing systems. This design environment has been given the acronym FIRST, standing for Fast Implementation of Real-time Signal Transforms. It consists of a design methodology together with a unified collection of CAD tools. It has been created to fulfil the objectives set out in chapter 1 using techniques identified in chapter 2. The underlying concepts and structure were originated by the author and P. B. Denyer. Software expertise was recruited through the Department of Computer Science at Edinburgh University, in the form of N. W. Bergmann, who worked at writing the first and second versions of the software, for six to nine months. These versions were written in IMP [1]. From this the author learned the necessary programming skills to maintain and extend the software thereafter. A third version of the software was translated from the original by J. Nash into the C language [2], with modifications to introduce dynamic storage and increased run time efficiency. Pilot studies of system design using FIRST were undertaken by the author and P. B. Denyer. System implementation studies have also been done by a number of visiting academic and industrial visitors. Latterly, this area of system designs has been greatly extended by S. G. Smith. Finally, the details of the test architecture were worked out mainly by A. F. Murray in

consultation with P. B. Denyer and the author. The author has contributed jointly in all areas of the design environment but principally in the areas of the methodology, the underlying architectural principles and the cell library.

3.2. Overview

Figure 3.1 shows the main components of the design environment which are as follows. Firstly, an underlying design methodology has been developed. Secondly, a hardware description language (HDL) has been defined. This provides a mechanism for precise design specification. Thirdly, an intermediate description code (IDC) has been defined. Fourthly there is a language compiler which compiles HDL to IDC. Fifthly, there is a physical design subsystem (PDS), which generates mask geometries from a compiled IDC. Sixthly, there is a behavioural description subsystem (BDS), which generates simulated behaviour corresponding to a compiled system from files of input signals. As a part of the design methodology, there is a strategy for built-in-testability and self test. This leads to the seventh component of the design environment, namely facilities for automatic test pattern generation (ATPG). Additionally, a number of peripheral pre-processors and post-processors have been developed. These take the form of short programs written by a variety of users to ease their task, and typically fulfil the function of specific signal generators, display packages and HDL generators. Finally, as a result of the unified automation environment, its HDL and the generated output files, documentation of system design is part of the actual design process.

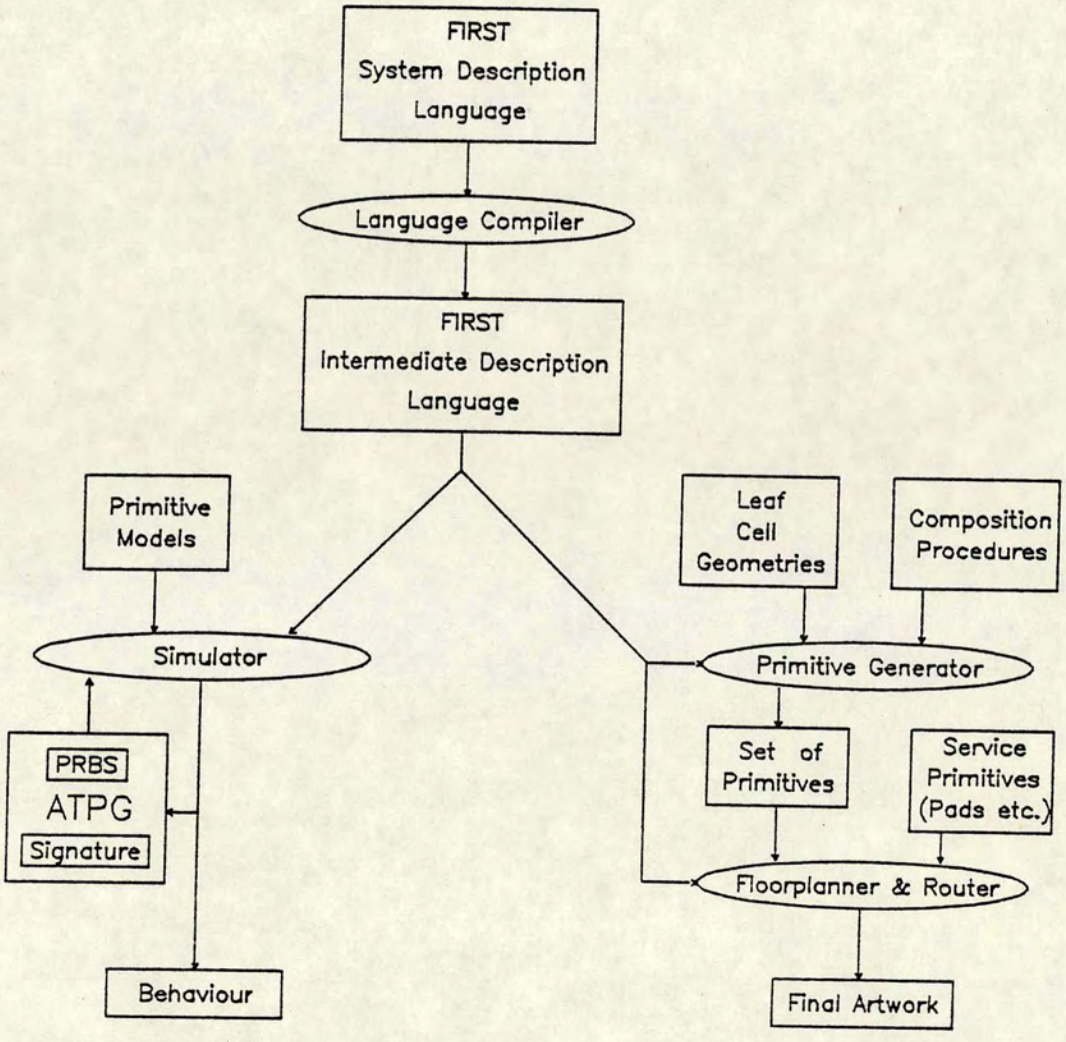

FIRST
System Description
Language
Language Compiler
FIRST
Intermediate Description
Language
Primitive
Models
Leaf
Cell
Geometries
Composition
Procedures
Simulator
Primitive Generator
PRBS
ATPG
Signature
Set of
Primitives
Service
Primitives
(Pads etc.)
Floorplanner & Router
Behaviour
Final Artwork

Figure 2.1

3.3. Architectural Basis

The key to the structural organisation of the design environment has been the adoption of a bit-serial architectural design methodology. This was proposed in [3]. Reasons for this choice were given in chapter 2. Some details of the form adapted and developed for this work and have been set out in [4]. The main impact of this approach has been on the implementation of the physical design subsystem (see section 3.8), on system partitioning, on the use of pipelining and functional parallelism and on communication.

3.4. Specification and System Design Capture

A system design specification, however formulated, consists of a functional component and a performance component. Design capture is the process of formulating this specification in a succession of forms which eventually translate into the actual system.

At a comparatively high level, the functional component of the specification can be completely defined by using a flow graph of interconnected functional elements. Systems engineers are accustomed to designing using this idiom. It translates very simply into a net list description, consisting of functional elements with inputs and outputs and nodes which define how the inputs and outputs are connected. The HDL defined in FIRST is simply a hierarchical version of such a net description language which also reflects the key physical boundaries (chip boundaries) of the system when implemented. The hierarchy allows arbitrary grouping and concealment of design detail.

Keywords and reserved symbols

OPERATOR	PIN		=	
CHIP	POUT		[	
SUBSYSTEM	PCIN		]	
SYSTEM		PCOUT		(
END			)	
	VDD		,	
SIGNAL		GND	;	
CONTROL		NC	->	
			<-	
PADIN		TIMES		+
PADOUT		THROUGH		-
PADORDER			*	
	CONTROLGENERATOR		/	
CONSTANT	EVENT		!	
WORDLENGTH		ENDCONTROLGENERATOR		:

ENDOFPROGRAM

Primitive names

ABSOLUTE	ADD	BITDELAY	
CBITDELAY	CONSTGEN	CWORDDELAY	
DPMULTIPLY	DSHIFT	FFORMAT1TO1	
FFORMAT2TO1	FFORMAT3TO1	FLIMIT	
FORMAT1TO2	MSHIFT	MULTIPLEX	
MULTIPLY	ORDER	SUBTRACT	
WORDDELAY			

Table 3.1

A FIRST HDL specification consists of a sequence of keywords, identifiers and constants together with arithmetic operators and various separator characters. Table 3.1 gives a listing of the keywords. Keywords are used to define the classes of objects comprising a system. These are nodes and functional elements. Nodes can be of type SIGNAL or of type CONTROL. The only predefined nodes are VDD, GND and NC, which are the supplies and not connected; all other nodes must be user defined. Functional elements are of type primitive, OPERATOR, CHIP, SUBSYSTEM or SYSTEM. Elements of type primitive are all predefined; they are given by the primitive name keywords. Elements of the other types are all user defined. The elements of type primitive, CHIP and SYSTEM,

have actual hardware counterparts. The elements of type OPERATOR and SUBSYSTEM reflect structuring within the specification. The general form for any functional element is:

name [plist] (clist) slist

where name is the user chosen, or predefined identifier, plist is a list of parameters, clist is a list of control inputs and outputs and slist is a list of signal inputs and outputs. If preceded by one of the keywords OPERATOR, CHIP, SUBSYSTEM or SYSTEM the above statement is a definition start. Definitions are terminated by the keyword END. Without one of these keyword prefixes the statement becomes a call or instantiation of the element. All user defined elements must be defined before they are called. The definitions of type CHIP and SYSTEM are both implicitly also calls. Constants and arithmetic expressions are used for parameterisation of functional elements, thus giving the facility to define classes of functional element. The parameters are evaluated on instantiation of a particular instance. This mechanism allows rapid modification of specifications to investigate quantisation effects etc., by the minimum alteration of an HDL specification. The HDL also supports a mechanism for condensed expression of long lists of nodes and repeated calls of functional elements (cascading).

The language is much simpler and more easily learned than conventional high level programming languages. It is, however, adequate for specifying a wide range of simple to complex digital signal processing systems. A formal definition and description of the language (version 2) is given in appendix I; details concerning the primitives are given in appendix II. The syntactic form of the language owes much to the influence of [5,6], as does the overall

software architecture.

It will be observed that the definition of function is explicitly captured in an HDL specification. The question remains as to how the performance component of the specification can be expressed. Primitives form an important group of keywords, to which there corresponds a defined function. However the definition of each primitive comprises more than simply its syntax and function. To each primitive there also corresponds an actual hardware circuit. As a result of the way that this has been designed it also possesses a known performance. Further, because of the way in which systems are synthesised from primitives the performance of any arbitrary synthesis is also determinate. As a result of these two factors any HDL specification will possess an implicit performance. Conversely, by adopting the design methodology and system mapping techniques presented in chapters 4 and 5, a specific performance can be built into the HDL specification.

In summary the HDL specification of a system captures its function explicitly as a net list of functional elements. The performance specification is derived from the primitives and the functional architecture. The architectural methodology and system mapping methods described in chapters 4 and 5 provide a means for translating a mathematical definition of function and a bandwidth definition of performance into an HDL specification. The HDL provides a level of description appropriate for compiler implementation. An example of how to describe a system using FIRST HDL is given in appendix III.

3.5. The FIRST HDL Compiler

The FIRST HDL provides a mechanism for exact design specification. The next step in complete design capture is to generate an intermediate code (IDC) from which both the behaviour and the physical circuit design can be generated. This IDC is an expanded but coded version of the HDL with the hierarchy removed. In addition the significant portions are factored out. These are the chip definitions, for passing to the physical design subsystem, and the system definition, for passing to the bound behavioural simulation subsystem. Compiler technology for accomplishing this task is well known. For reasons of initial convenience, the implementation was based on SKIMP [7], a simple compiler designed for teaching the principles of compiler writing. The principles involved can also be found in [8]. The structure of the compiler is shown in the outline pseudo-code given next.

```
main {
     declarations
     decode options and set flags
     read in and reduce syntax definitions
     read in and store predefined primitives
     open files for IDC output and diagnostics
     open input HDL file
     while( !EOF) {
          process_statement from input
     }
}
```

This pseudo-code, if expanded, refined and translated into a programming language can be made to implement the required function. The routine process_statement expands as shown next.

```
process_statement {
     read statement, clean up and reconstruct
     perform lexical analysis
     perform recursive descent analysis of syntax
     generate IDC
}
```

The input to the first phase is HDL; the output is cleaned, abbreviated HDL. This is passed to the lexical analysis phase, which recognises keywords, names and constants and inserts a coded record of these into a lexical array and hash table. This record is taken by the syntax analyser, which performs an analysis of it with respect to the table of reduced syntax definitions. The analysis routine is called recursively, and attempts to find a match between the record and the definitions. If successful it generates a condensed analysis record in the form of a linearised analysis tree. The analysis record is interpreted by the code generator, which then outputs appropriate IDC for each item.

The run time performance of the compiler is satisfactory, typical systems can be compiled almost instantaneously. For this reason the simple approach has been adopted and sophisticated optimisation schemes eschewed.

The entry of new primitives into the compiler is trivial, requiring only the addition of the corresponding keyword to the file of predefined primitives. The format requires also information on the number of parameters, control inputs, control outputs, signal inputs and signal outputs. The entry of new syntax is not so straightforward and requires a detailed knowledge of the compiler implementation. This is mainly related to the code generation phase. Addition of new syntax is easily incorporated into line reconstruction, lexical and syntax analysis.

3.6. Primitives and the Primitive Cell Library

Primitives constitute the kernel of the design environment. This is so because the approach to system synthesis is based on assembling functional elements from primitives. As a preview of what follows it is worth summarising the definitions associated with each primitive at the various levels at which a primitive can incarnate within the design environment. In the HDL and IDC, each primitive is defined as a keyword and syntax. In the cell library, each primitive has a dissociated definition in the form of the component leaf cells required to construct all instances of it. Primitives also possess definition as a composition template (see below). This is formulated, in the physical design subsystem, as a parameterised procedural definition (cp., section 3.8). The introduction of a new primitive into the design environment involves definition at each of these levels and in the correct format for incorporation. A major issue associated with the design of the primitives is that of verification. Verification is achieved by establishing the consistency of these definitions with respect to each other. This topic is addressed in section 3.10.

The approach to defining primitives has been dominated by the requirement to meet a guaranteed operating performance, through which the system performance can be ensured. In order to achieve this, it was decided that primitives should be composed from custom designed component circuits, or leaf cells. Custom design can guarantee both performance and local optimisation for each such cell.

The generation of primitives from such custom designed leaf

cells is achieved by means of an architectural template. This is a general pattern for the assembly of a set of leaf cells to generate a class of similar circuits. The general pattern is expressed in terms of the particular leaf cells used, in terms of a set of one or more parameters and in terms of the arrangement of the leaf cells. Each such class of circuits constitutes a primitive. Each case of the pattern is a particular instance of the primitive. These general patterns will be termed composition templates.

The assembly of the leaf cells according to such templates must be governed by an internal set of conventions and defined interfaces. These are local and internal to the primitive and need only be sufficient to ensure that, within the generality of the composition template, any assembly of cells will satisfy the required performance. Further each primitive must have an external interface. It must also satisfy a set of geometric and electrical conventions to ensure that communication between primitives satisfies the functional and performance requirements set down. The creation of these design conventions and interfaces ensures that, within any system, arbitrary assemblies of primitives will satisfy both function and performance.

The design of primitives is also constrained by the need to fit a global assembly strategy. The latter is an architectural template for chip generation and constitutes a class of floor plans which the physical design subsystem can produce. The adoption of such an approach has advantages with respect to performance and local optimisation and also in simplifying the software implementation of a physical design subsystem. However, it also has disadvantages. Firstly, iteration on the design of leaf cells, in the

light of global assembly requirements, is excluded. It was adjudged, however, that the sophistication required to achieve successful implementation of such a technique for primitive generation was beyond the scope of this research. The second disadvantage is that custom designed leaf cells have to be technology specific. The significance of this can be understood in the light of the rapid development of new technologies and the need to re-implement existing leaf cell libraries in such technologies. The custom design effort required will be considerable. Again a solution to this problem lies outwith the scope of this research, but urgently requires attention.

The technology chosen for leaf cell design was the so called standard five micron polysilicon gate nMOS process. This choice was dictated by the availability of such a process through the SERC supported Edinburgh Micro-fabrication Facility (EMF). The geometric design rules adopted were a set of generalised geometric design rules given in [9]. The electrical design rules were taken from the EMF design rules [10]. For this process an operating clock frequency of 8 MHz should be attainable.

Circuit design techniques were based on two phase, non-overlapping-clocked, dynamic, ratioed logic design [11,12]. Ratio conventions of 4:1 and 8:1 were adhered to for rectangular transistors and a ratio of 10:1 or 12:1 where `snaked´ or `hooked´ transistor misalignment could be critical. To avoid race hazards, every stage in the pipeline should start with the appropriate clocked pass transistor followed by an inverter, or gate and then any further pass or gate transistor logic. Phi1 clocked stages must alternate with Phi2 clocked stages. It is worth mentioning

that although pass transistor networks may appear preferable from the point of view of area and power dissipation, they do inflict reduction on test coverage.

Using these circuit techniques, together with the design conventions outlined below,leaf cells were iteratively designed using circuit simulation to make up the cell library. To avoid confusion it should be stated that this process of iteration occurs only during leaf cell design; it does not constitute a part of the physical design subsystem. Leaf cells, once designed and verified are fixed objects.

3.7. Interfaces and Architectural Templates: Foundations for a Physical Design Subsystem

Interface definition is a mechanism for ensuring that the required performance of primitives, chips and systems is attained. There are three important classes of interface: those between leaf cells, those between primitives, and those between chips. In order to ensure performance the following conventions are used.

Internal to primitives, each leaf cell is designed so that it is capable of driving (at the required fraction of the minimum operating frequency of 8 MHz) any of the possible outputs which may be connected to it as defined by the composition template. Usually this is done by worst case design and the task is much simplified by the features of modularity, local communications, and pipelined operation exhibited by bit-serial circuits.

Between primitives, a common standard interface buffer is used. This is associated with primitive outputs. It has been

designed to drive worst case loads of 10 pF, (which is equivalent to maximum wire lengths with a fanout of five). This buffer dissipates approximately 3 mW and has an area of approximately 0.02 sq mm. Again the sparseness of bit-serial communication makes the use of such an interface a feasible proposition. The operation of this buffer uses one half bit cycle (or one clock phase) within the operational timing of the primitive. For convenience and efficiency, there are inverting and non-inverting versions of this buffer. Signal and control are latched by phi2 at the input to a primitive and are also valid through the output buffers during phi2. The buffers are made available during primitive leaf cell design, for incorporation into the circuits. In this way a primitive's capacity to operate the buffers correctly can be verified. Figure 3.2 shows the circuit schematic for a non-inverting buffer. An actual buffer can be seen in Figure 3.3, which shows a micro photograph of a section of a chip including a buffer in the field of view.

Between chips the interface is pipelined to take one full bit time to transfer one bit. This is necessary because inter-chip loads can be in the range 20-30 pF due to pad, lead-to-lead and wire-to-wire capacitances. Output pads latch their signals using phi2 and input pads latch their signals using phi1. Output pads require a very large driver. Input pads use the common-standard on chip buffer common to all primitives. The public domain Xerox-Parc [13], pad designs were modified slightly to suit.

The overall consequences of these interface definitions is that a mechanism exists for ensuring performance. It should be noted that one side effect of the chip-to-chip interface definition

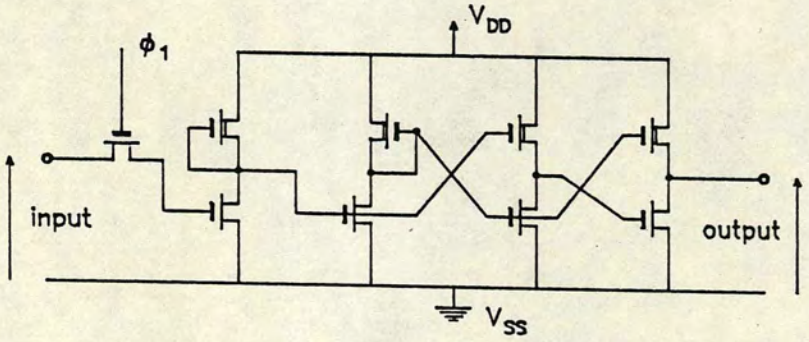


Figure 2.2

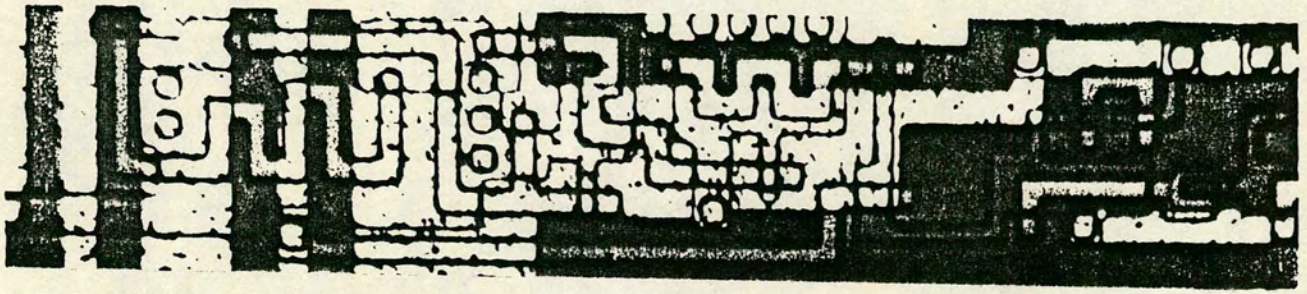


Figure 2.3

is that there is a one bit latency overhead associated with partitioning.

Architectural templates constitute the basis for a mechanism whereby the physical design subsystem can carry out its function of generating chip mask geometries from a compiled IDC. They consist of a set of geometrical patterns and conventions which govern the design of primitives, the assembly of primitives and the synthesis of chips. Because the context in which a leaf cell or primitive is to be placed partly conditions the best design for that leaf cell or primitive it is necessary to outline architectural templates for chips then primitive and finally leaf cells. Global context is important in determining a local configuration, especially when the local configuration cannot be subsequently changed to suit its surroundings.

The concept of synthesising systems from primitives suggests a general composition strategy, in which primitives are maintained as tight physical clumps of layout, to be wired together according to the system design. This obvious approach is nevertheless a powerful one, and its advantages are worthy of examination. Most importantly, the high functional level of the primitives secures a low connection-to-layout ratio. All of the internal connections between gates and transistors are compact, and externally invisible. This is not the case for example with gate-array and standard-cell techniques, where the `atoms´ are at the gate level only. In that case all functions must be reduced to the gate level, where inflexible positioning results in an inefficient wiring scheme. By permitting custom primitive layout advantage can be taken of any implied regularity and structure in, say a multiplier

or delay element. Also a degree of local optimisation can be applied in designing such elements. This leaves only the relatively sparse interconnection of operands and results between primitives. Here the bit-serial architecture works to advantage, for these signals are generally single wires only.

The architectural template for chips is a generalised floorplan. The datapath (Mead and Conway, 1980, Sherburne et al., 1981, Batali et al., 1981, Schrobe, 1981) is an excellent example of such a generalised floorplan for bit-parallel architectures. A successful alternative architecture, again with bit-parallel organisation, has been proposed by Siskind et al., (1981) where the need for constrained architectures is also recognised. Several generalised floorplans are feasible for bit-serial systems. The generalised floorplan described next is a simple but appropriate one for 4 to 6 micron nMOS technologies, using the circuit techniques outlined above.

This floorplan style, shown in Figure 3.4, will be termed `Manhattan Skyline´ for the similarity it bears to a cluster of skyscrapers and reflections in the water. It comprises a central communication channel, flanked by two regions for the placement of bit-serial primitives. Signal routing is implemented through the central channel only; there is no connection between neighbour primitives except, possibly, through the central wiring channel at their base. Thus primitives communicate by receiving and transmitting data via the channel across the `waterfront´. Chip input and output signals are routed to peripheral pads via the ends of the channel. Note that, for bit-serial architectures, the communication core rarely dominates the chip area.

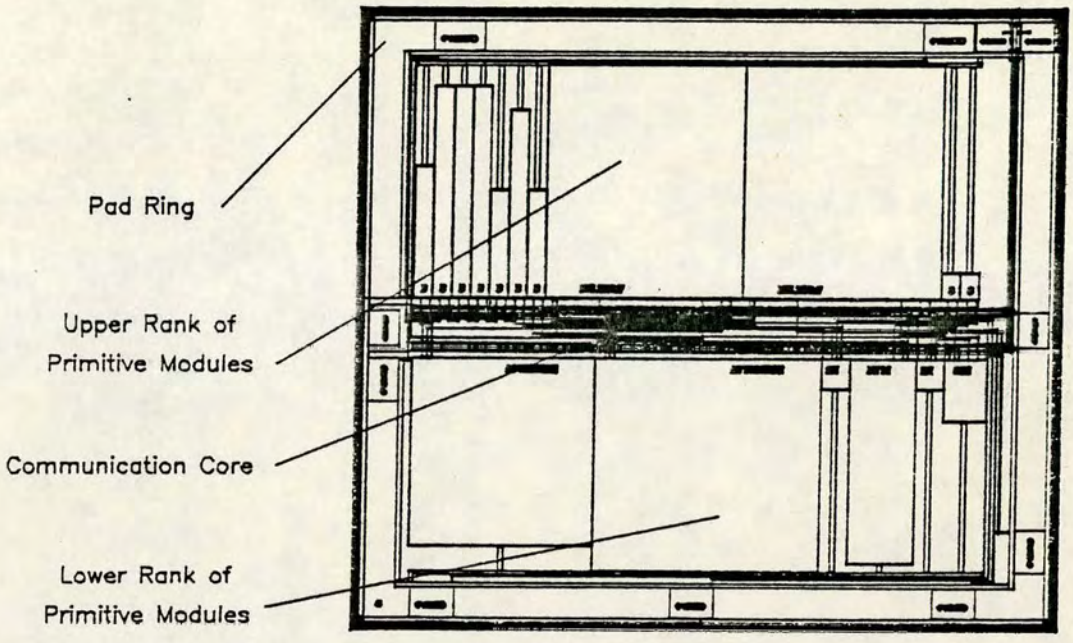


Figure 3.4

The simplicity of this floorplan style belies some important advantages. Signals in the communication channel are for the most part routed in metal, beginning and ending in short stubs of diffusion (or polysilicon). This is a useful feature where the technology offers only one low-impedance, low-capacitance interconnection medium (in this case a single level of metal). The single channel is easy to route, and primitive ordering, to minimise chip area, can also be computed quickly.

The provision of common services (global power and clock busses) throughout the chip must now be considered. There are two well defined sub-tasks: servicing the primitives, and servicing the pad ring. To do this service ducts are provided along the `water-

front' for the primitives, and a service ring around the pad channel, as shown in Figure 3.5. This principle is easily extended to arbitrary floorplan conventions if the primitive service duct is allowed to become a flexible `umbilical cord', routed around arbitrary

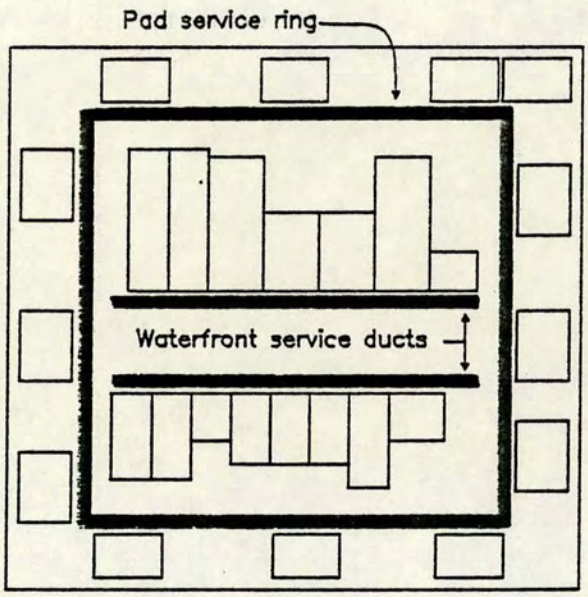


Figure 3.5

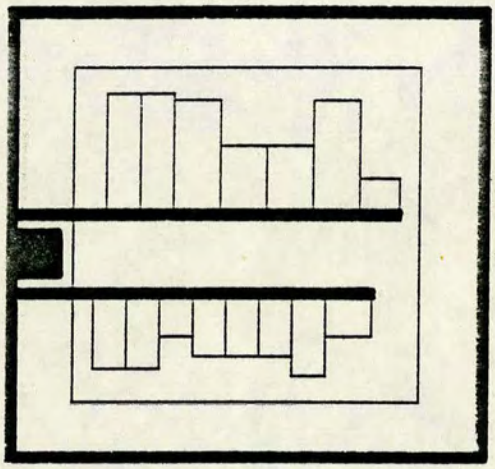
primitive placements.

One problem with such an arrangement is the desirability of maintaining low impedance power and clock busses as far as possible. Normally this means maintaining these busses in metal for the most part. There is no problem when more than one metal layer is available, for then power and clock lines can be crossed without resorting to a relatively high impedance diffusion or polysilicon

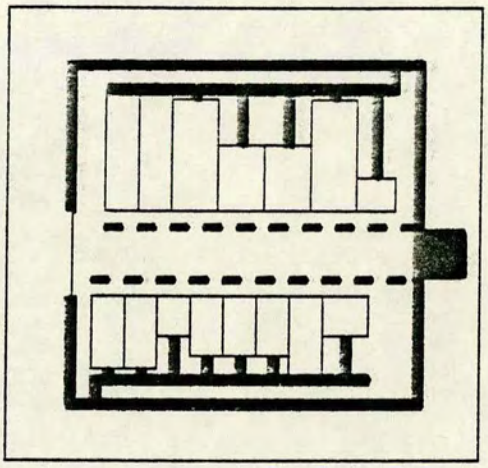
crossunder. However, when only one metal layer is available, as is the case in the process being used, it becomes impossible to supply power from the duct to an adjacent primitive without using at least one crossunder. the solution is to split one of the power rails (VSS say) away from the duct and route it separately on the opposite side of the primitives. As shown in Figure 3.6 it then becomes possible to route power to all primitives, and cells within primitives, without crossunders. A similar solution exists for the pad channel, supplying VSS and VDD from opposite sides of the ring. Crossunders are needed for clock distribution, but their number and length are kept to the absolute minimum.

Figure 3.6 shows the complete service routing schema for FIRST based on these principles. It includes fixed positions for each of the supply pads (at opposite ends of the communication core), and for the clock pads, phi1 and phi2. Without altering this scheme it is possible to replace the clock pad pair by an on-chip clock generator if this is preferred. Long runs of parallel pairs of clock lines are separated by a power supply line as a precaution against noise and cross talk. Pad names in the diffusion layer can be output below this ring, adjacent to each pad, for bonding identification.

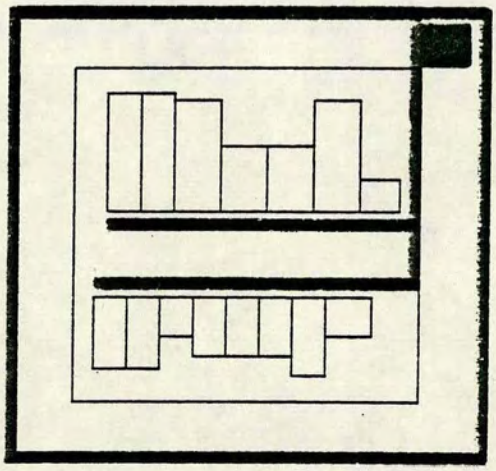
In addition to its role as a power bus, VSS is also sometimes used as a source for signals of `zero´ value (000...00). Thus a secondary VSS bus is also routed within the waterfront duct for this purpose. It has a higher impedance than the main power busses (because of cross-unders) and so may not generally be used as a power source. A convenient exception to this rule is the use of this bus to partially power input and output buffers to primitives



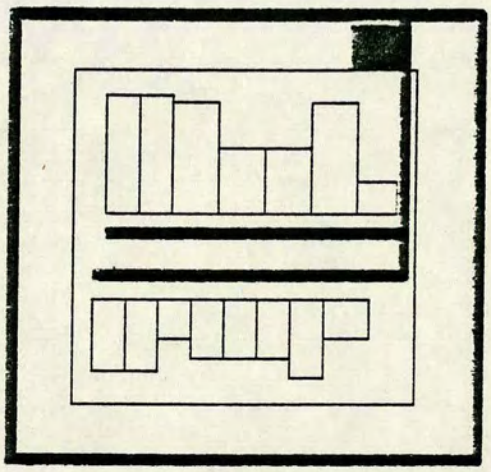
VDD distribution



VSS distribution



ϕ_1 distribution



ϕ_2 distribution

Figure 3.6

which span this service duct, as discussed below. The crossunders are designed to cope with this function.

An architectural template for primitives can now be defined in relation to the chip generalised floorplan. There are no absolute restrictions on primitive width or height. The factors which must be taken into consideration are as follows. An attempt is made to minimise the waterfront dimension. This is done so as to maximise the number of primitives that can be conveniently placed in one chip. (In practice, chips generated using the proposed floor plan tend to have aspect ratios (length to width) greater than one whereas fabrication and packaging conventions usually demand a close to square aspect ratio.) All data inputs and outputs are restricted to occur along the waterfront. To aid channel routing, the allowable positions of i/o ports are quantised to be at even integer multiples of the wire pitch of the vertical wiring medium. (This will allow vertical wires from opposing rows of primitives to interdigitate.) In this case the standard adopted was a wire width of 4 lambda, with a space of 3 lambda, giving a total wire pitch of 7 lambda in the diffusion layer. Thus primitive i/o ports are positioned on a permissible grid of twice this pitch (14 lambda), commencing from coordinate 0 lambda on the extreme left. This is illustrated in Figure 3.7.

It is convenient to permit primitive layout to spread under the service duct for the special cases of the input and output buffers, so saving layout area. The use of these buffers, which

---

Lambda is a unit of linear dimension and represents a convenient smallest unit of length (cp. Mead and Conway, 1980). Its absolute value depends upon the final choice of technology for implementation.

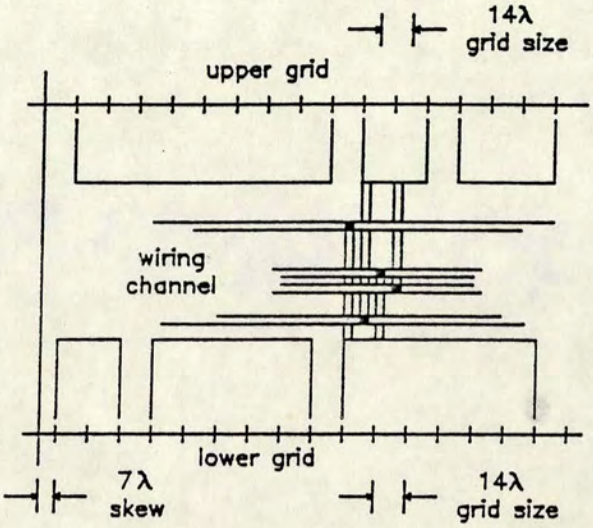


Figure 3.7

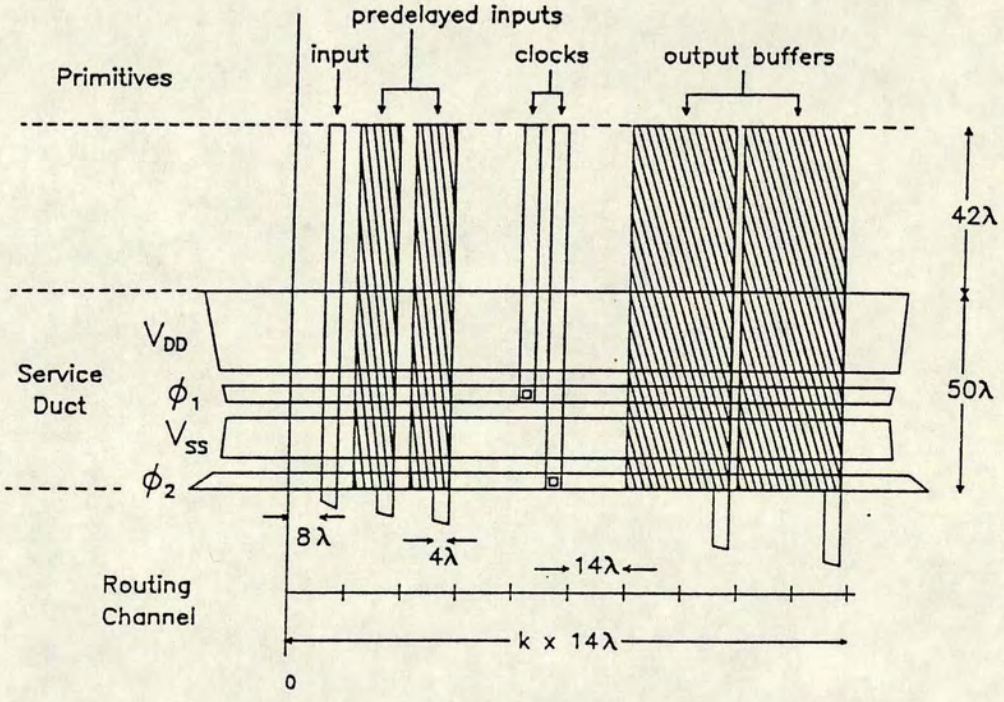


Figure 3.8

take a standard form, is illustrated in Figure 3.8. The output buffer is the final drive stage attached to every primitive output. The input buffer is optional on any primitive signal input port. Its function is to provide an optional delay of one whole clock cycle (bit time). This is an expedient facility for handling inputs to primitives that have become misaligned by one bit in time. The incorporation of a predelay cell at a primitive input comes for free (in terms of layout area) in this way, instead of requiring an extra primitive and wiring, for the sake of a single bit of delay. The incorporation of predelay appears as an option on signal input ports for all primitives, except bitdelay and control bitdelay.

The lower bound on primitive `waterfront´ width is determined by the number of inputs and outputs. Associated with each input there is a width of 14 lambda, and with each output a width of 28 lambda. Where possible, the primitive width should be made equal to this lower bound. The primitive can gain access to its VDD supply through any output buffer. Access to VSS is on the `skyline´, where a bar of metal the full width of the primitive should be placed. Each input buffer must have access to VSS from the primitive. Finally, primitives should be minimum area layouts, subject to the above constraints.

An architectural template for leaf cells can now be defined. The only requirement is that it be compatible with the architectural template for primitives. The VSS and VDD net for each primitive should be interdigitated and should be constituted entirely on the metal layer having no crossunders. The requirement for I/O on the same edge, as well as for modular composition and local

communication internal to each primitive requires that a linear flow configuration be bent back on itself. The general linear form is usually made up of input buffers, followed by functional logic blocks, followed by an optional cascade of added delay followed finally by output buffers. This can be bisected and bent back so that input and output attach to the same edge of the layout. The final layout of leaf cells has then to be tailored in relation to their sizes and positions relative to adjacent cells. In order to minimise primitive area such arrangements may have to be different for differing configurations of a primitive (i.e., for the various parameter values). This leads to conditional selection of leaf cells. For entirely modular structures, different parameter values may only require a different number of repetitions of the same cells. The leaf cells should be designed so as to form simple tessellations to make up primitives, as well as meet the functional, performance and minimum area criteria. Global and broadcast schemes are to be avoided, and modular pipelined communication used instead. Beyond these, leaf cell design is unrestricted so as to benefit from custom design optimisation.

3.8. The Physical Design Subsystem

The physical design subsystem (PDS) is a set of routines which take compiled IDC and generate chip mask geometries. These correspond to the function defined by the IDC, and have a performance fixed by the primitives and process technology. The routines automate the architectural templates and interface conventions reviewed in the previous section and use them as a means to assemble chip geometries. The programming basis for the PDS is an embedded language (originally this was ILAP [14]). This allows

parameterised procedural definition of symbols (which represent groups of geometric shapes) within a high level programming language (cp., chapter 2).

The structure of the PDS is shown in the following piece of pseudo-code.

```
!================================================
     main {
          open I/P and O/P files
          init_layout()
          read_pre_def()
          read_idc()
          place()
          wire_up()
          layout()
          output_stats()
     }
!================================================
```

The function of init_layout() is to initialise ILAP and define global symbols. The function of read_pre_def() is to read in the primitive definitions and build up the corresponding data structures needed. The function of read_idc() is to read in the compiled IDC system definition, check its node list (this includes checking for fan-out, contention, floating and redundant nodes) and construct the necessary data structures. This routine includes calls to the composition routines and outputs CIF definitions as well as computing the actual sizes of the primitives needed. The routine place() computes the positions in which to locate the primitives so as to achieve a minimum value for the sum of the areas of the two regions on either side of the wiring channel. The routine wire_up() serves to generate the data necessary for making all the connections between primitive input and output ports. The routine layout() uses the previously created data structures to generate the full chip geometry. This routine outputs the remaining CIF

definitions. Finally the routine output_stats() prints out various chip statistics.

Thus, the four principal routines are read_idc(), place(), wire_up() and lay_out(). The routine read_idc() invokes primitive generation routines, one call corresponding to each distinct primitive required. As an example of a primitive composition routine, consider the assembly of the bit-serial multiplier primitive, which the author designed for the cell library. This particular multiplier consists of a cascade of cells, each of which handles the product associated with a pair of bits from the coefficient word. These cells are held as leaf cells of layout in CIF code. The beginning and end cells in the string are special; the intermediate cells are of one type which is repeated as necessary to build a multiplier of the required coefficient word length. In order to communicate legally with the central wiring channel, the otherwise linear cascade of cells is doubled over at (or close to) the halfway point, and an additional interconnect cell is used as a link at the folding point. An assembly plan is shown in Figure 3.9 for the case of a multiplier with an odd number of main cells.

Given that the sizes of the leaf cells are known (from layout), the following outline for the composition routine, when expanded, may be used to assemble the multiplier as a function of one parameter - the coefficient wordlength:

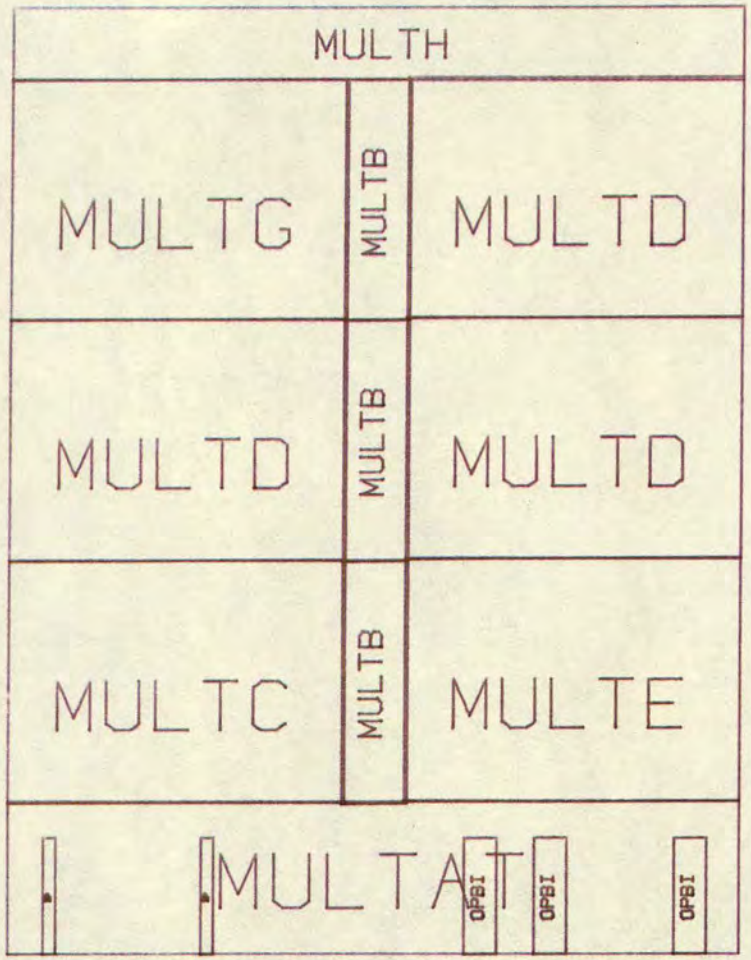


Figure 3.9

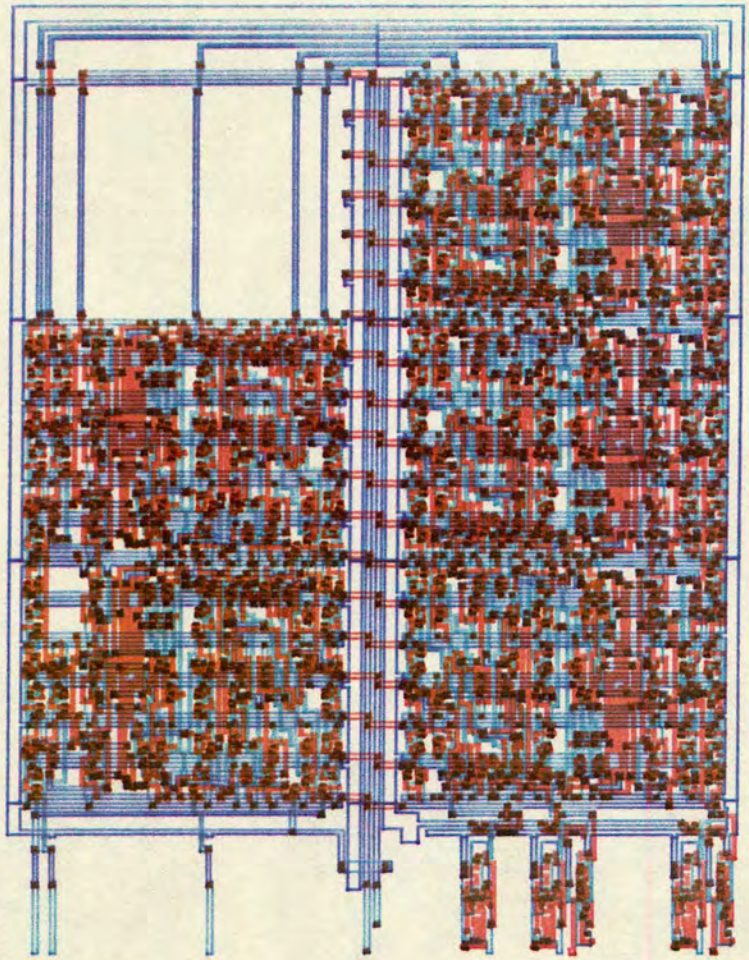


Figure 3.10

```
!============================================================
multiply()
{
     declarations
     retrieve parameter values for this instance
     calculate internal constants
     check parameter values
     compose symbol identifier (name)

     if(!symbol_exists(name)) {
          symbol(ttname)
          draw("MULTAT")
          draw I/O buffers
          draw("MULTB")
          draw("MULTC")
          draw("MULTE")
          for(k = 1;k <= limitb;++k)
               draw("MULTB")
          for(k = 1;k <= limitdt;++k)
               draw("MULTD")
          if((coeffbits-4) % 4 != 0)
               draw("MULTG")
          else if(coeffbits > 4)
               draw("MULTD")
          draw("MULTH")
          for(k = limitdb;k >= 1;--k)
               draw("MULTD")
          endsymbol()
     }
     evaluate and return height and width
     evaluate and return I/O port relative positions

}
!============================================================
```

where the following routines are examples of calls to the embedded language and have the function described. The routine draw() outputs the CIF shapes of the symbol named as its argument; this is done so as to have the correct position and orientation also. Note that suitable values for position and orientation must be passed to the draw() function and must subsequently be updated for the next call. At this level the details are suppressed for clarity. The function symbol() is a routine to generate a CIF definition start and endsymbol() to generate a CIF definition end. The function symbol_exists() is a function which interrogates the data structure generated so far to see whether the symbol has already been defined

or not; it returns TRUE or FALSE. The pseudo-code given above can be expanded directly into an appropriate high level language function or routine to define the multiply primitive. An example of full layout generated by such a routine is shown in Figure 3.10.

This block of layout, representing the full primitive module, is then available to the subsequent routines of the Physical Design Subsystem of the silicon compiler for placement and routing. There is no restriction to the use of a single parameter, many of the primitives have two or three parameters.

The next task is concerned with determining a suitable arrangement for the primitives thus generated. This is done by the place() routine. Placement of the primitives within the floor-plan proceeds so as to minimise the sum of the upper and lower regional areas. The factors which determine total area are as follows:

the height of the tallest block on the top row

the height of the tallest block on the bottom row

the length of the longer row

the thickness of the wiring channel

The first three factors depend only on which modules are placed on the top and bottom rows respectively. To find the best arrangement, the primitives are first arranged in order of descending height, are all allocated to the upper region and the resulting area is evaluated. One after another the primitives are moved from the upper region to the lower region in order of ascending height, to try to detect an arrangement with smaller total area. The primitives allocated to the lower region are arranged in order of increasing width. Trial perturbations are then carried out; primitives are moved, in order of ascending width, from the lower region

to the upper region to even up the total lengths of each region. Further trial perturbations are made to determine whether the relocation of any single primitive from the upper to the lower region will reduce the total area. When the arrangement with least area has been identified, its allocation of primitives to the upper and lower regions is recorded in the data structure.

It is recognised that this algorithm may not converge on the global minimum value for chip area. This is because, the area occupied by the wiring channel has been neglected, in evaluating the total placement area. The trade-off is between fast implementation and simplicity against increased area. It was estimated that the loss of area was likely to account for approximately only ten percent. Further, a general minimum area solution to the place and route problem is not known and the time that would be required to implement a more sophisticated algorithm was unlikely to be offset by the gain.

The actual arrangement of primitives, within their gross allocation to upper or lower regions, is done according to their relative groupings in the compiled IDC. This reflects the ordering given by the designer in his HDL description. Designers tend to write this description in a manner that follows the general flow of information in the system. Such a strategy then leads to closely coupled modules being close together on the chip, with a resultant minimisation of the total wiring net. Again, more sophisticated placement algorithms based jointly on the area and interconnect could be developed.

The upper and lower rows of primitives will be located in

opposition, to either side of the designated wiring channel, and one row is shifted by a single wire pitch (7 lambda). This allows i/o ports from the two rows to interdigitate across the entire width of the channel without having to keep account of clashes (coincident wires of different nodes). A simple channel router can be used, with metal wires running horizontally and diffusion connections running vertically. Clearly this scheme would be well suited to any two layer process, for example double-layer metal. The top and bottom row offset ensures that connections can be made between any two ports with a single horizontal wire and two vertical wires, that is without doglegs. Figure 3.11 shows a typical wiring channel section.

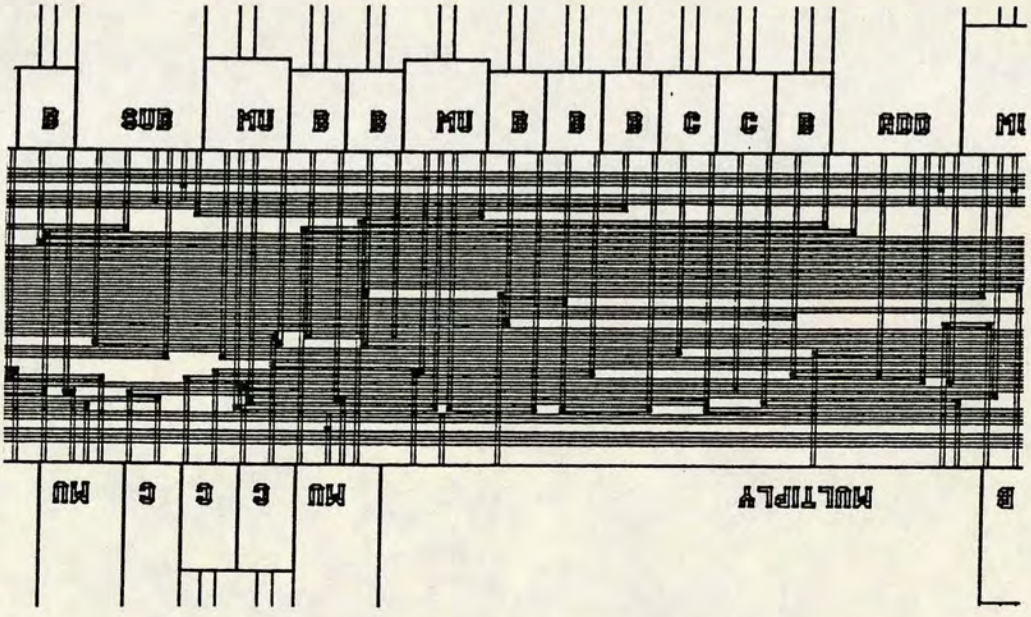


Figure 3.11

Given the ordering of primitives, as determined by the place() routine, the routine wire_up() computes the necessary arrangement

of wiring within the channel so as to connect up ports which have the same node number. This is done by sorting the ports according to two keys, firstly their node number and secondly their position. Ports with the same node number are to be connected. For each node an allocation to slots (the parallel locations in the channel where the wires are laid out) within the channel is made. A running account of occupied and vacant locations in each slot is kept so that clashes can be avoided, connections to the same nodes merged and so that the wiring channel can be compressed as much as possible. The termination positions for metal and diffusion tracks and the positions of contacts are recorded into the data structure, for use by the layout routine.

On completion of place() and wire_up(), the required positional arrangements for primitives and wires has been generated and the process of producing the actual geometric shapes can be commenced. This is the function of the layout() routine, which sequentially generates the parts of the chip as indicated in the following piece of pseudo-code.

```
!=============================================
layout() {
     define_name_pads()
     define_corners()
     symbol()
          layout_operators()
          layout_metal()
          layout_diffusion()
          layout_contacts()
          calculate_boundaries()
          layout_ground()
          layout_vdd()
          layout_clock()
          layout_pads()
          layout_clockpads()
          layout_bias_gen()
          print_size()
     end_symbol()
}
!=============================================
```

The template for pad location and routing is as follows. The pads are all either signal or control input or output pads, to be routed from the ends of the central wiring channel. Having fixed the positions of the supply and clock pads (along with an area for the substrate bias generator, if used) three segments are partitioned off within the pad channel. To allow some control over the eventual device pinout, the user must assign groups of pads to each segment, and to further specify their order within these segments. The relevant pad primitives (input or output) are then assembled and located with approximately even spacing within each segment. The pad routing is then implemented by a simple bus arrangement, from the lower and right hand pad channels to one end of the communication core, and from the opposing edges to the other end of the core. The junction-box shown in Figure 3.12 completes the connections between the communication core and the pad bus. Note that it is necessary for the central channel router to have brought any input and output signals to the appropriate end of the channel. More detailed ordering of these signals is not necessary because a

good interchange facility is available through the junction box. An improvement is possible, if subsequent to calculating this arrangement, it is compacted. Once again, the approach taken trades off simplicity against the degree of optimisation possible.

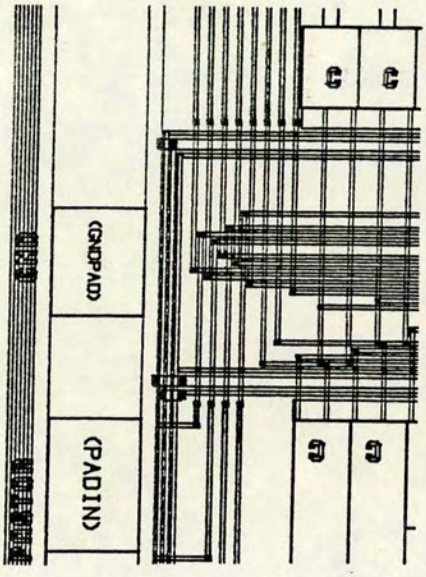


Figure 3.12

Finally chip statistics are evaluated and output. Currently these include only chip size. However, on completion of primitive characterisation, accurate estimates of power dissipation could also be added.

3.9. The Behavioural Description Subsystem

The behavioural description subsystem consists of a set of (verified) computational models for all primitives together with a simulator kernel. The purpose it serves within the design environment is to give the designer a bit-level emulation of any system

which he specifies in FIRST HDL. In this way, firstly, coding errors in the HDL can be eliminated. Secondly, the system behaviour for any set of inputs can be examined, both at the outputs and at any internal node. This allows a designer to obtain debugging information for modifying the system HDL specification, if this is needed. Additionally, it allows the designer to evaluate the system, with respect to such features as system word-length, quantisation noise etc. Finally the BDS provides a basis from which test pattern outputs can be generated.

It is argued that this addition to a structured design methodology overcomes the shortcomings of a purely structural design method as advocated by Mead & Conway in [11] which, on its own, does not in fact offer effective tools for the complete elimination of design bugs, or for testing and automatic test pattern generation.

In order for the behavioural design subsystem to capture the necessary aspects of the design it must be capable of representing all idioms expressed in the HDL. Thus it is driven by a compiled IDC description, which contains the system design. Further, in terms of detailed fidelity, the BDS must preserve accuracy right down to the bit level.

One approach to building such a BDS is to work entirely at the bit-level. Primitives are modelled by the boolean processes they perform, and the state of the entire system network is computed from these at every clock cycle. This can be thought of as being roughly equivalent to a switch level simulation of the entire system. In practice primitive models at this level are undesirably

complex. Also the simulation of detailed processes within every primitive leads to prohibitive simulation times, and storage requirements. Furthermore, these internal processes are of no direct relevance to the system designer, who is only concerned with activity, particularly word-level behaviour at the network level. Words correspond to signal samples and this is the finest level of resolution of interest. However, a guarantee of isomorphism between switch-level, bit-level and word-level is needed. This suggests the need for hierarchical simulation with automatic translation between levels and proven isomorphism. In particular, it becomes apparent that an appropriate level at which to implement the BDS is that of word level models. Preliminary experiments with simulation run times at bit and word level [15], confirmed this. The problem of isomorphism between levels is then relegated to the task of verification, which is discussed in a later section.

In order to outline the organisation, structure and functioning of the BDS it is necessary to describe how the signal values associated with a node may be represented at the word level. In the hardware, each node of a system specified using FIRST has a bit serial stream of values, 0 or 1 , appearing on it. Each primitive processes the serial inputs together with the primitive's internal state to give the resulting delayed output bit stream(s). Values on nodes and within primitives are clock driven. This pipelined, bit serial operation of primitives, occurs concurrently in the hardware realisation. In order to simulate it efficiently using a conventional, operation-serial von-Neumann machine, word level models are extracted as follows. Each bit stream which appears on a signal node of the system has associated with it a

control (C1 or CLSB) bit stream which marks the occurrence of the least significant bit of each new word (cp.: [4]). CLSB is high for one bit, during the occurrence of the LSB of the associated signal(s), and is low at all other times. Thus there is a natural grouping of bits into words. These words represent the signal values on the node at the interval during which they occur. Define the word on a node to be the word composed from the bit on the node at CLSB high together with all subsequent bits on the node until the next CLSB high, but not including the bit then on the node. The bits are packed LSB first, in other words right justified.

This definition implies that words are only to be associated with a node at discrete time intervals and in between there is no word on the node. This notion of a word on a node allows a natural definition of events associated with nodes and hence to the construction of an event driven simulator model, in place of the previously required clock driven model. In turn this leads to a reduction of the computation load required of the simulator. (The ambiguity between the use of the term event here (and in the rest of this section) and in the context of the HDL controlgenerator should not cause any problems.)

An event has associated with it a time, a node, and a value. An event may be said to occur on a node only at the time when the associated CLSB pulse is high, and the value associated with the event will be the word, as defined above, on the node at that time.

The bitwise interpretation of words relates the higher level description to the actual bit streams observable on nodes as follows. At CLSB high the LSB of the word appears on the node; this

is followed by each subsequent bit of the word, equally separated throughout the interval until the next CLSB high.

Events on the input nodes of primitives must be synchronised, i.e., they must occur simultaneously. The simulator checks throughout operation for this and warns if it detects synchronisation failure. If inputs are not synchronised then it is because there is a mismatch between the latencies in the paths of the inputs and has been caused by a design fault. This allows the designer to check and detect such faults easily. (Note that when such warnings are given the behaviour of the simulator is not necessarily the behaviour of the hardware if it were fabricated, but both would be faulty.)

When an event occurs at the inputs to a primitive there are associated words, or signal values at those inputs. This means that, from a word level model of the primitive, output values from the primitive can be computed. The output time, node and value can then be stored, in association, as another event.

Thus a simple event driven simulator can be structured around a set of utilities for reading in IDC code, for constructing the necessary data arrays and hashing indices, for setting input and output, for checking and warning etc., and for managing event scheduling, together with a set of word level models for all the primitives. The utilities constitute the simulator kernel and the primitive models form a library of descriptions, or definitions with which the kernel works. The simulator is event driven and simulates the operation of the system and its components on a word by word basis.

An event on any node invokes, in turn, models of all primitives having that node as an input. New values are then computed for each of the output nodes to `occur´ at the appropriate time (namely the time of the input event plus the primitive latency). The scheduling of events is handled by storing all pending events in a queue. All events due at a given time are removed from the queue, and the new output words are then computed and inserted into the queue at appropriate times. Thus further events (with associated time, node and value) can be scheduled.

Inputs and outputs to the system are via external files. These data files are essentially a list of events associated with the system input or output nodes. The format requires time and datum to be given in a file for each input node. Output data is produced as a time, node, datum format. It is not difficult to write programs which accept this format and produce alternative representations from it thus creating user-friendly interfaces. For example, input data may be generated from a set of software `signal generators´ that produce commonly-used waveforms (weighted sums of sinusoids, square, triangular and saw-tooth waves, step functions, chirps etc) under user controlled selection and parameter value choice. Equally, output data may be interpreted in a graphic form, producing oscilloscope-like traces of signal activity on nodes. In this way it is possible to construct a productive work-bench emulation environment for the systems designer.

Figure 3.13 shows simulation output of this type for a complex-to-magnitude chip. The inputs are two orthogonal sinusoids of equal but decaying amplitude, which should produce a decay envelope at the output. The simulator verifies this behaviour for

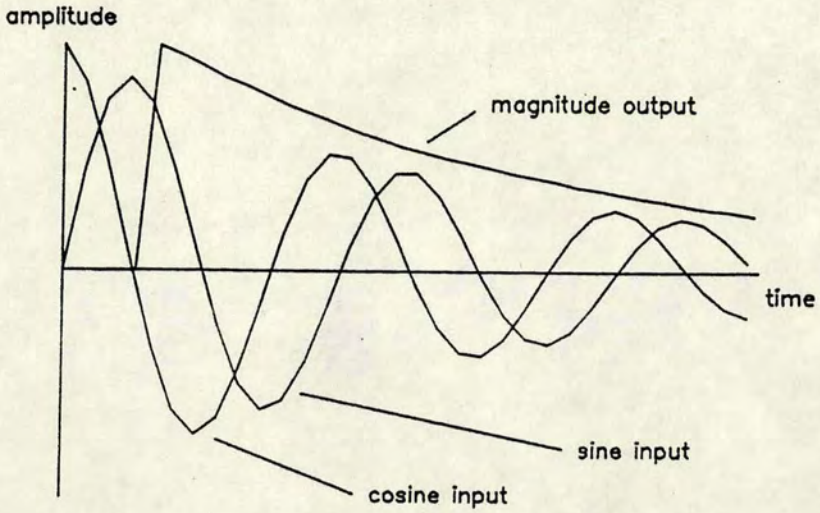


Figure 3.13

a run of 32 data samples over several cycles of the sinusoids. Note some initially invalid activity at the output as the pipeline is cleared. This is equivalent to the latency of the operator.

It remains now to show how the primitive definitions are formulated so as to interface with the simulator kernel. The simulator model of a primitive is based on utilities to interface it to the simulator kernal and on its word-level function (addition, multiplication, etc.), together with appropriate adjustments to maintain bit-level fidelity. This type of model is illustrated in the following pseudo-code description of the MULTIPLY primitive:

```
!=====================================================================
multiply()
{
     declarations
     retrieve control node number for this instance
     if(time of event on control node == time) {

          retrieve signal node numbers for this instance
          retrieve parameters for this instance
          check parameter values, warn if wrong
          compute latency and any other constants required

          if(time of event on any data node != present time){
               timing_warning() and diagnostics
          }

          retrieve input signal values
          check formats, warn if out of range
          multiply input values to required degree of precision
          format bytes of product for output
          enter output values, event times and nodes on queue
     }
     else
          timing warning and diagnostics
}
!=====================================================================
```

There is also a facility for storing and retrieving any primitive-associated, internal-state information which may be needed.

3.10. Correctness and verification

The stated aim is to produce a functionally correct system at the first attempt. It is unacceptable that time and money should be spent on fabricating a VLSI system that is anything other than correct. This holds especially for VLSI as a development medium for prototype or low-volume requirements. Conventional approaches to VLSI synthesis are distinctly hit-and-miss in this respect. Often as much resource is spent on verification as on the design process itself, and just as often the design will not function correctly at the first attempt. The general cause of this is attributable to the complexity of the VLSI design process. There may be many levels of representation, with scope for error in

interpreting between each of them. The silicon compiler can avoid this situation by virtue of `correctness by construction´.

There are two approaches to ensuring that chip designs are correct. One is to attempt a design and then test it, either by circuit extraction and simulation, or in the limit, by fabrication. If the result of the simulation (test) is unsatisfactory, the design is modified and this process is iterated until the simulated (tested) behaviour is correct. The cycle is illustrated in Figure 3.14. The scheme becomes practically unwieldy for VLSI systems because the data structures involved become extremely large, and the computational demands of whole-system simulation are prohibitive, if even possible. A more acceptable variant of this approach is to design hierarchically and support the resulting structure with a series of extractor/simulators which verify behaviour between levels. This technique can be made to work in practice for the correct design of complex circuits, but it is clearly costly to implement, and the many levels and iterations are demanding on expert skills, time and computational power.

The alternative approach is to use only techniques of construction that are known to be correct, and to complement these by automating their implementation and verifying this automation. In this way, synthesis can be made automatic, rapid and correct. To ensure correctness therefore it is necessary to verify the techniques and their implementation. For this, there are three areas which require attention. These are the area of software verification, the second is the area of the underlying techniques and assumptions and the third is the area associated with the primitive cell library. This leads to formal systems for specification and

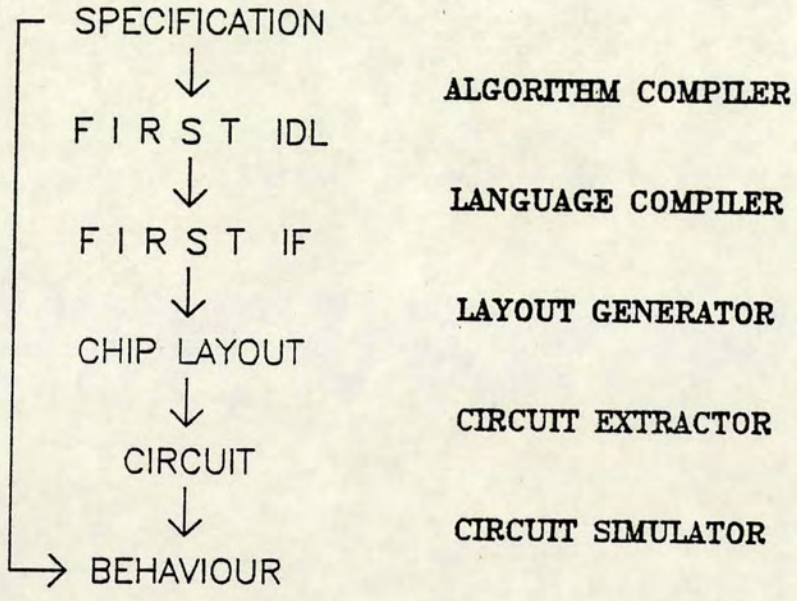


Figure 3.14

verification. Research into applying these techniques has been started [16,17,18,19], but lies outwith the scope of this thesis. Such approaches to circuit design will, however be of great importance to the development of the next generation of design methodologies and CAD tools.

A model of the structure of FIRST is shown in Figure 3.15. The requirement to verify the behaviour of each system (as above) is replaced by previously verifying a set of equivalences between the components of the physical design subsystem and the models and assumptions built into the simulator. These include:

the primitive leaf cell layouts

the primitive composition routines

the assumed interface conventions

the chip wiring routines

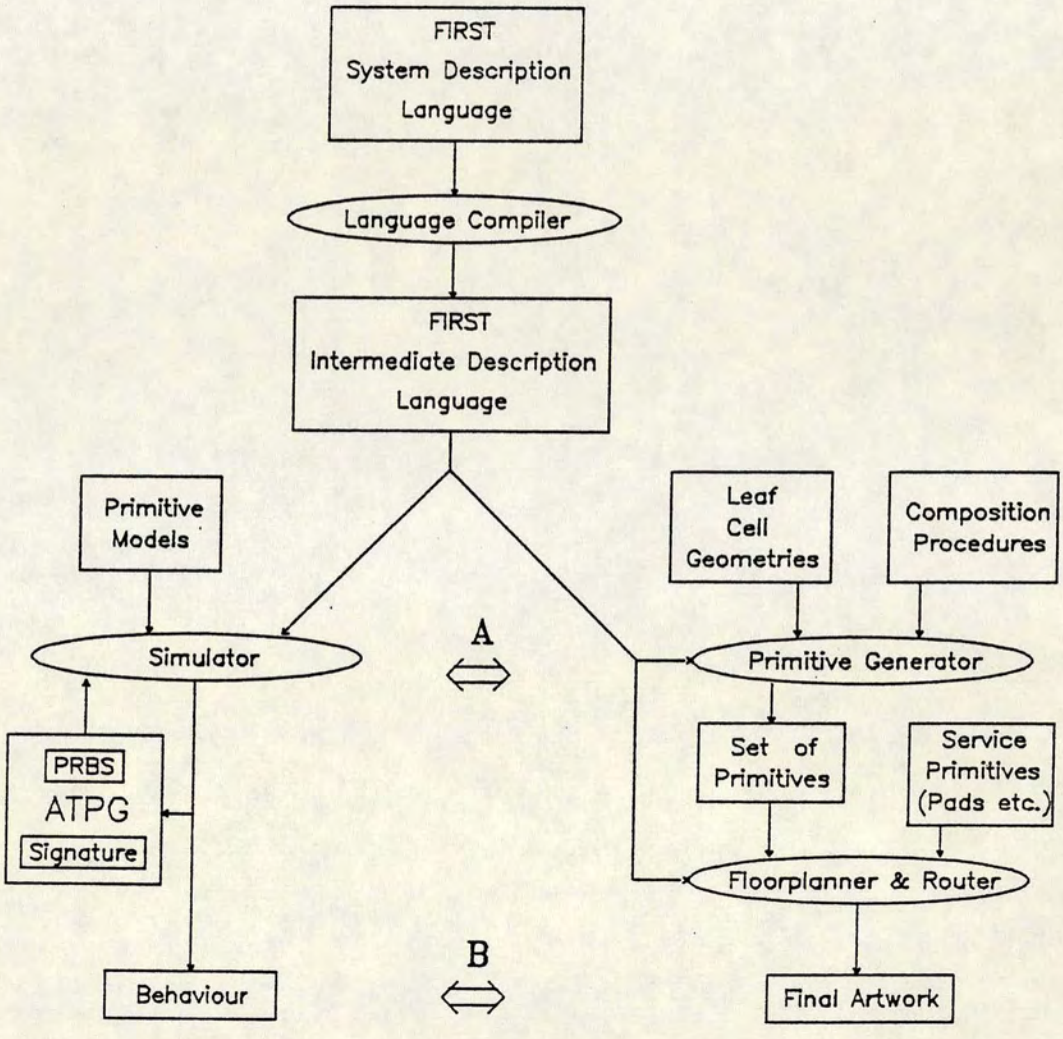

FIRST
System Description
Language
Language Compiler
FIRST
Intermediate Description
Language
Primitive
Models
Leaf
Cell
Geometries
Composition
Procedures
Simulator
A
Primitive Generator
PRBS
ATPG
Signature
Set of
Primitives
Service
Primitives
(Pads etc.)
Floorplanner & Router
B
Behaviour
Final Artwork

Figure 3.15

the behavioural primitive models.

Once the software and the data structures are validated, there is an assurance that each system that is composed by them will be valid. Validation of the primitive library is of particular importance, since this component is not fixed, as are the compiler, layout generator, and simulator. It is capable of expansion as required, or there may exist different versions for different technologies.

Briefly, this validation is carried out using a benchmark test set. This contains examples of each primitive in alternative configurations, to test different cases of the composition routines as well as the leaf cells. An HDL description of this test set is compiled. Circuit extraction and simulation are then used to validate the primitives, their interface conventions, and the channel routing. A final proof is given by a fabrication-test run imposed on any primitive before acceptance into the `public' library. Figure 3.16 is a microphotograph of such a primitive verification chip. Further details may be found in chapter 7.

3.11. Design for Test, ATPG and Self-Test

A substantial proportion of system development cost is absorbed by testing. As system complexity increases, the test problem increases exponentially, unless specific steps are taken to redress this effect. Test can be divided into design testing, parametric testing, initial product testing, in-service testing and reliability testing. The problem of testing is to provide a means of distinguishing good from faulty components and systems, at minimum cost. This is achieved by the design of test input pat-

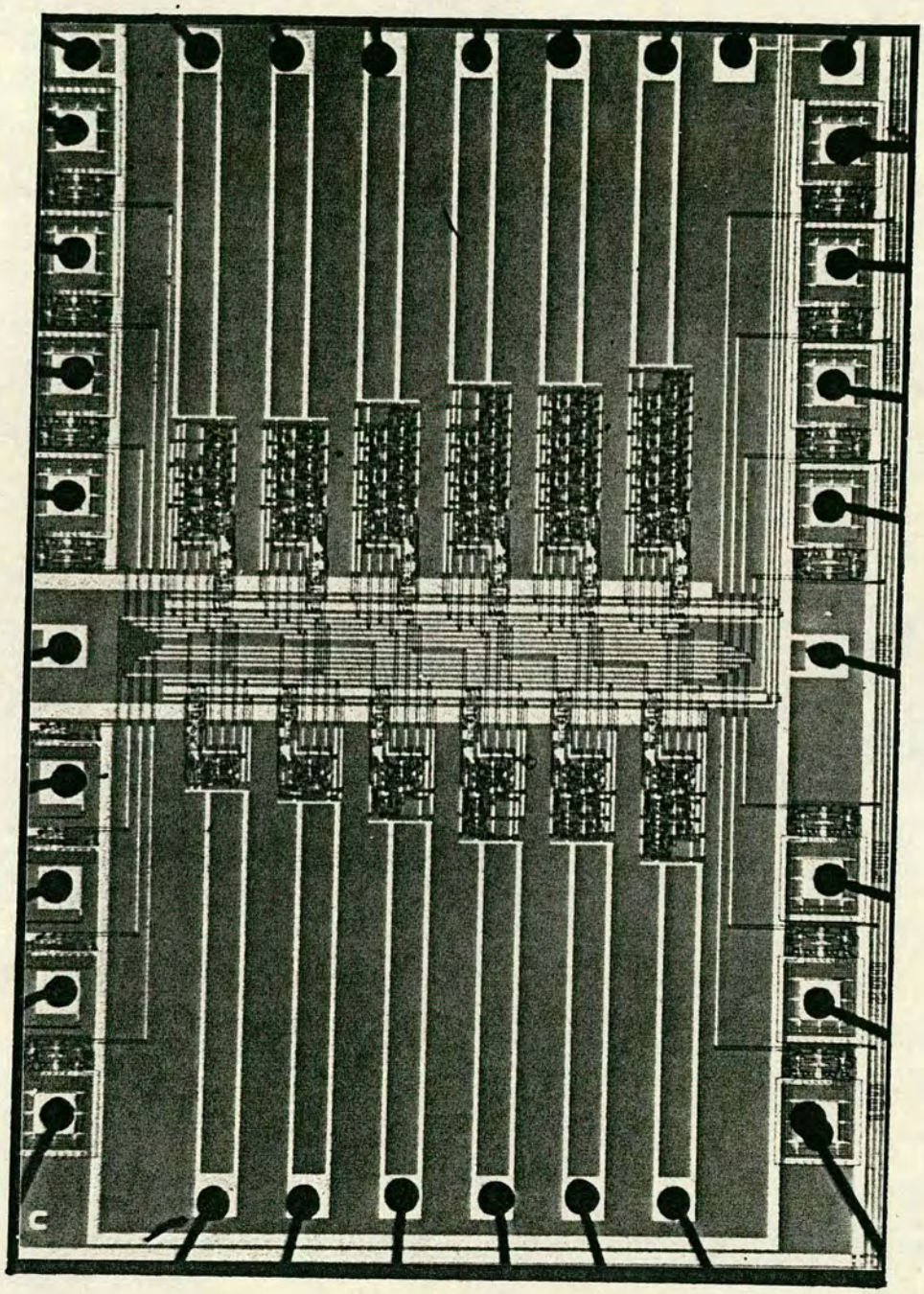


Figure 3.16

terns (TIPs) which exercise the system and reveal, by means of the test output patterns (TOPs), the presence of any internal faults in the system. The extent to which internal faults can be detected for any given TIP is termed the test coverage of the TIP with respect to the system under test. The usual approach to TIP design is first of all to create fault models which are injected into a simulation model of the system. TIPs are then devised and run through the simulation and measurements of fault coverage are made against the resulting TOPs. Here again, as system size increases, the simulations required grow exponentially. This rapidly leads to systems which are untestable, either because not sufficient information about the internal conditions can be deduced from TOPs, or because to do so would require test times far in excess of the total system lifetime. Thus techniques have been devised in order to reduce test complexity and prevent the creation of such systems. These techniques involve reducing internal combinatorial and sequential depth by structuring system design from the outset with testability in mind. A review of established methods can be found in [20,21,22].

In the context of the design environment under consideration the requirements are as follows. Firstly, system synthesis guidelines are needed so that it is not possible to create HDL descriptions which compile into untestable silicon structures. Secondly, an automatic test pattern generator is required, which can give an arbitrary but known test cover for any system under design. Within this environment parametric testing and reliability testing are not addressed since the resources to support investigation in these areas is not available.

In order to make an appropriate choice of test strategy and so as to extract system design guidelines, it is necessary to examine the properties of bit-serial primitives and of networks composed of such. The features of importance for testability are as follows. Firstly the fan-in and fan-out of internal nodes is low; what is more, this is equally true for primitives as it is for systems. Secondly the combinational and sequential depth of all primitives is small. Fourthly, there are very few primitives which possess resistance to random pattern testing. (This occurs when nodes are not affected by random pattern test inputs.) Fifthly, nearly all primitives possess the property that they propagate random patterns. Factors one, two and three contribute generally to easing all forms of test. Factors four and five make bit-serial primitives, and any architectures synthesised from bit-serial primitives, particularly amenable to random pattern testing. The choice of random pattern testing is made on the grounds of economy of TIP generation and the extent of test coverage available. Detailed development of the arguments is set out in [23,24,25] In summary, it has been established that short (1023 bit) pseudo-random bit sequences (PRBS) can give one hundred per cent test cover for even the most complicated primitives (against the single stuck at fault model). Further, it has been shown that, as a result of propagation of randomness, networks of such primitives can be tested with the same degree of cover by the same sequences. Furthermore, full system simulation to establish the degree of test cover is not necessary.

There are only two consequences for system design at the level of HDL description and above. Firstly, all loops must be broken so

that the sequential depth is kept to a minimum. This allows internal states of such loops to be controlled and observed. The mechanism for breaking loops is to introduce a multiplex point with one external input and also to tap a node as output. Secondly, any primitives which do not propagate a PRBS must be isolated from the network during test. Again this may be done by the introduction of appropriate multiplex points.

The problem of ATPG now reduces to generating suitable length independent PRBS TIPs. This can easily be done using a computer program. The corresponding TOPs can be derived by injecting these PRBS TIPs as input to a BDS system simulation of the system HDL. The length of test sequences and the degree of test cover can be established from the primitive test characteristics and the principle of propagation of randomness.

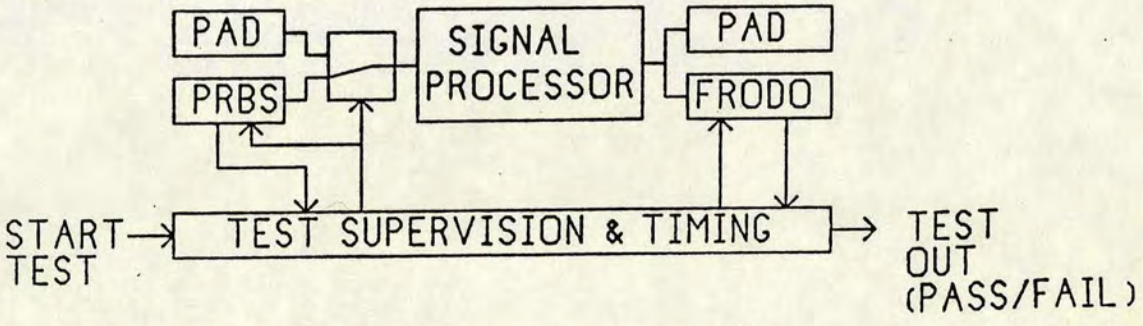


Figure 3.17

A consequence of this approach to system test is that it can also be implemented in hardware at minimal cost. PRBS generators can be implemented as small circuits. Further, data compression techniques (signature analysis) and recognition of test-pass or

test-fail can also be built into a system in hardware. This gives rise to an integrated, self-test capability. The architecture for achieving this is illustrated in Figure 3.17. A PRBS register and multiplex point is associated with each input pad. A feed-back register observer of data output (FRODO) is associated with each output pad. Since bit serial systems are not pad limited there is ample area for both in the pad ring. Further, the timing and control circuitry required is small and so can also be located in the pad ring. This results in self-test with no increase in chip area and an increase of the order of one per cent in active area. By appropriate configuration of the multiplex points, the architecture can be set up for individual chip test, for system test (including interconnect) and for system run-time configuration (cp. Figure 3.18). The achievable fault coverage when using signature analysis is reduced below 100%, but can be made arbitrarily high by increasing the length of the PRBS TIPs. Typically 1023 bit PRBS can give greater than 99% coverage of single stuck at faults.

3.12. Summary

This chapter has described a design environment comprising a design methodology, techniques for design automation and a collection of integrated software tools, for implementing bit-serial signal processing systems. The author has taken a major part in the specification, development and implementation of this design environment. In the next two chapters a generalised technique for mapping mathematical functions into HDL descriptions is given. This is followed by an evaluation of what can be achieved using the design environment and a review of hardware verification of the primitives.

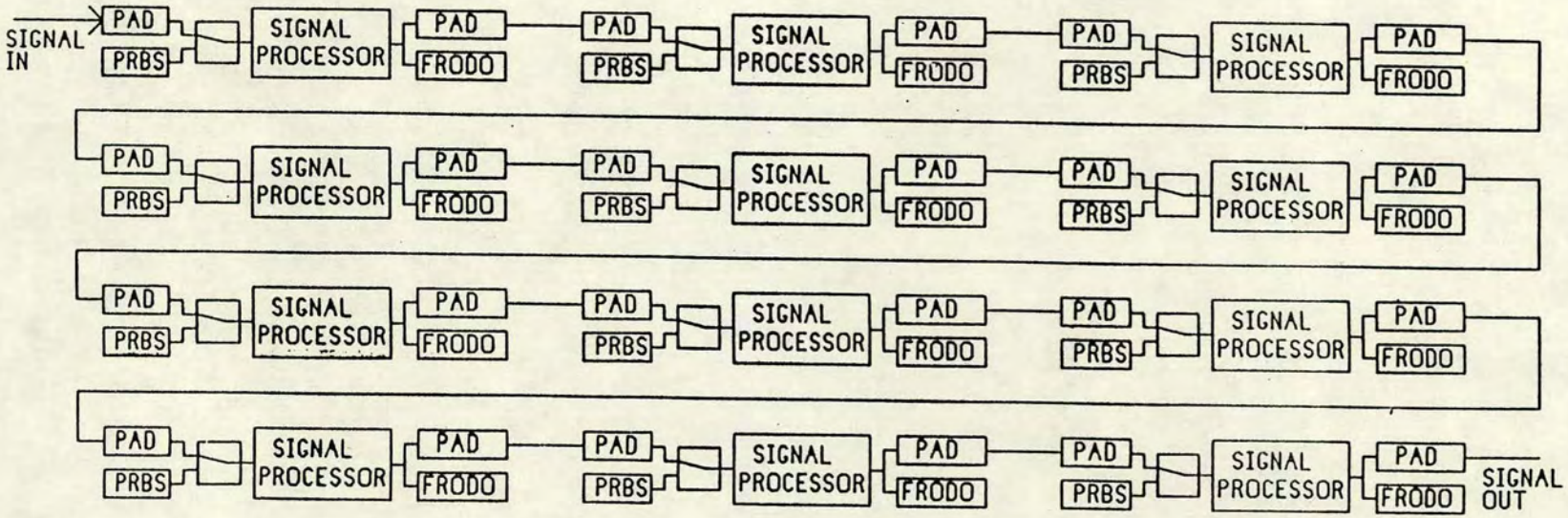


Figure 3.18

References

1. P. S. Robertson, "The IMP-77 Language," Department of Computer Science Internal Report, University of Edinburgh (1977).

2. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice Hall (1978).

3. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," pp. 131-140 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

4. P. B. Denyer, "An Introduction to Bit-Serial Architectures for VLSI Signal Processing," pp. 225-241 in VLSI architecture, ed. B. Randell, P. C. Treleaven,Prentice Hall (1983).

5. J. P. Gray, I. Buchanan, and P. S. Robertson, "Designing Gate Arrays Using a Silicon Compiler," Proceedings, 19-th Design Automation Conference, pp. 377-383 (1982).

6. J. P. Gray, I. Buchanan, and P. S. Robertson, "Controlling VLSI Complexity Using a High-Level Language for Design Description," Proceedings, International Conference on Computer Design, (1983).

7. D. J. Rees, "SKIMP Mk II," Department of Computer Science Internal Report, University of Edinburgh (1980).

8. A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison Wesley (1977).

9. R. F. Lyon, "Simplified Design Rules for VLSI Layouts," Lambda Vol. 2(1) pp. 54-59 (1-st Quarter 1980).

10. A. M. Gundlach and R. Holwill, Edinburgh Microfabrication Facility Design Rules: N-Channel Silicon Gate Process 2 (Revision B). March 1981.

11. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley (1980).

12. J. Mavor, M. A. Jack, and P. B. Denyer, Introduction to MOS LSI Design, Addison Wesley (1982).

13. J. Newkirk and R. Mathews, The VLSI Designer's Library, Addison-Wesley (1983).

14. J. G. Hughes, "VLSI Design Tools," Internal Report, University of Edinburgh Department of Computer Science (1981).

15. N. W. Bergmann, "A Case Study of the FIRST Silicon Compiler," pp. 413-430 in 3-rd Caltech Conference on VLSI, ed. R. Bryant,Computer Science Press (1983).

16. M. Gordon, "A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness," Department of Computer Science Internal Report, University of Edinburgh (1981).

17. G. J. Milne, "CIRCAL A Calculus for Circuit Description," Department of Computer Science Internal Report, University of Edinburgh (1982).

18. G. J. Milne, "A Simple Silicon Compiler and its Correctness," in Proc. 6-th International Symposium on Computer Hardware Description Languages and their Applications, ed. T. Uehara and M. Barbacci,North-Holland Publ. Co. (May, 1983).

19. G. J. Milne, "CIRCAL A Calculus for Circuit Description," Integration Vol. 1(2&3) pp. 121-160 (October 1983).

20. M. T. M. Rene Segers, "The Impact of Testing on VLSI Design Methods," IEEE Journal of Solid-State Circuits Vol. SC-17(3) pp. 481-486 (June, 1982).

21. T. W. Williams, "Design for Testability," NATO A.S.I., CAD for VLSI Circuits, Sijthoff & Noordhoff, (1981).

22. G. Grassl, "Design for Testability," NATO Advanced Study Course on VLSI Design, (1980).

23. A. F. Murray, "On the Effectiveness of Random Pattern Self Test for Bit-Serial Signal Processors," IEEE Trans. Comp., (to be published).

24. A. F. Murray, P. B. Denyer, and D. Renshaw, "Self-Testing in Bit-Serial Parts: High Coverage at Low Cost," Proc. IEEE International Test Conf., pp. 260 - 268 (Philadelphia, October 1983).

25. P. B. Denyer and D. Renshaw, VLSI Signal Processing: A Bit-Serial Approach, Addison Wesley (to be published).

CHAPTER 4

ARCHITECTURAL DESIGN TECHNIQUES

4.1. Introduction

The preceding chapter has described tools and techniques to assist in the production of sets of VLSI chip designs from functional flow graphs. A successful implementation now depends on the formulation of a flow graph that correctly represents the function. System architectures are at issue here, especially with regard to the provision and use of concurrency. In this chapter some system architectures for real-time signal processing are examined. Ways are developed by which to derive both an appropriate computation scheme and a corresponding hardware architecture for a general class of signal processing algorithms.

The material presented in this chapter has been developed by the author and has similarities to the work of other researchers [1,2,3,4,5]. It differs, however, in one significant aspect, namely in the treatment of bandwidth matching and multiplexed architectures. In this respect it incorporates and extends work done by Lyon [6]. A revised form of this chapter is to be found in [7], which also contains further examples of system case studies.

The approach is to formulate function as a recurrence or set of recurrences. In order to compute each output sample the operations of the recurrence are repeated a fixed number of times. Each repetition is termed an iteration and the operations carried out in each iteration are identical. A processor is defined to implement the operations required for a single iteration of the recurrence.

The major issue of concern is concurrency and its responsible application. It is necessary to achieve specific real-time sample rates using adequate but not excessive arithmetic processing resources. The range of possibilities (in terms of processors per iteration) is as follows. Firstly, the system can use only one processor, as in a von Neumann machine. This type of architecture is termed fully serial. Secondly, several processors can be used , but in any case fewer in number than there are iterations. Architectures of this sort are termed multiplexed architectures. Thirdly, the system can use the same number of processors as there are iterations. These are termed full array architectures. An example of such an architecture is the systolic array [8,9,10,11,12,13,14]. Note that the term array is used here to signify a linear array; however, the ideas may readily be extended to two dimensional arrays. Finally, for very high speed applications, more processors than there are iterations can be used. These could be termed distributed highly parallel array architectures.

The specific area of real-time applications addressed can be implemented using multiplexed architectures. Thus, the area between the fully serial architecture and the full array architectures is of primary concern. Generally, neither of these extremes is optimal for any given real-time application and the most efficient choice is a multiplexed architecture.

This chapter has a generality which extends beyond specific bit-serial constructions. The architectures could be applied equally to bit-parallel or other implementations with different timing and communication protocols (e.g., self-timed etc).

However, it should be observed that the design environment developed so far essentially reduces the design task to that of synthesising a high level hardware network to implement particular signal processing functions, that is of mapping function via computation scheme into an architecture; the rest of design has been automated. It is possible to take an entirely ad hoc approach to this task and then use a behavioural simulator of the type already described in chapter 3 for verification and correction. However, such an approach is error prone and time consuming. This chapter proposes a systematic way of mapping function into architecture for a class of functions and formulates architectures in such a way that they can be expressed conveniently and concisely as a hardware signal flow graph. The next chapter is concerned with the subsequent detailed issues of developing hardware flow-graphs specific to bit-serial implementations and the tools described in the previous chapter.

The material developed here, together with an algorithm specification language, could form a basis from which to develop a next generation of silicon compiler with a higher, algorithmic level of input specification.

4.2. An Example.

By way of an introduction, and as an example of the type of problem to be tackled, consider the implementation of a programmable transversal filter, sometimes referred to as a non-recursive finite impulse response (FIR) filter. The definition of this type of filter can be formulated in terms of the z-transform [Rabiner & Gold] as :

$$H(z) = b_1 + b_2 z^{-1} + \ldots + b_M z^{-M+1} \qquad 4.1$$

Equivalently the filter can be specified in terms of of the following equation, which expresses the filter output in terms of its previous inputs and tap weights :

$$y_i = b_1 x_i + b_2 x_{i-1} + \ldots + b_M x_{i-M+1} \qquad 4.2$$

The principal problem of implementation is to derive a computation scheme and a corresponding architecture which can eventually be expressed as a hardware flow-graph. For this particular problem there are many possible solutions, some of which are illustrated and discussed at the end of this chapter.

Equation 5. 2 can be reformulated as a recurrence since its evaluation depends on repeated multiply and add operations; these are the operations of a single iteration of the recurrence, which a processor must implement. From the hardware cycle time required to compute a single iteration and the input sampling frequency it is possible to determine how many such elementary operations can be carried out in one unit of sample time and therefore how many processors will be needed for the complete recurrence evaluation in real-time. This corresponds to finding a computation scheme which has the correct bandwidth for the application so that the corresponding hardware will be capable of, but not greatly exceed, the required throughput rate. This is termed bandwidth matching.

Suppose that an application requires a 256-point filter. This filter can be implemented at the one extreme with 256 multipliers and 256 adders etc; this would be a full array architecture (see Figure 4.1(a)). At the other extreme it can be implemented with

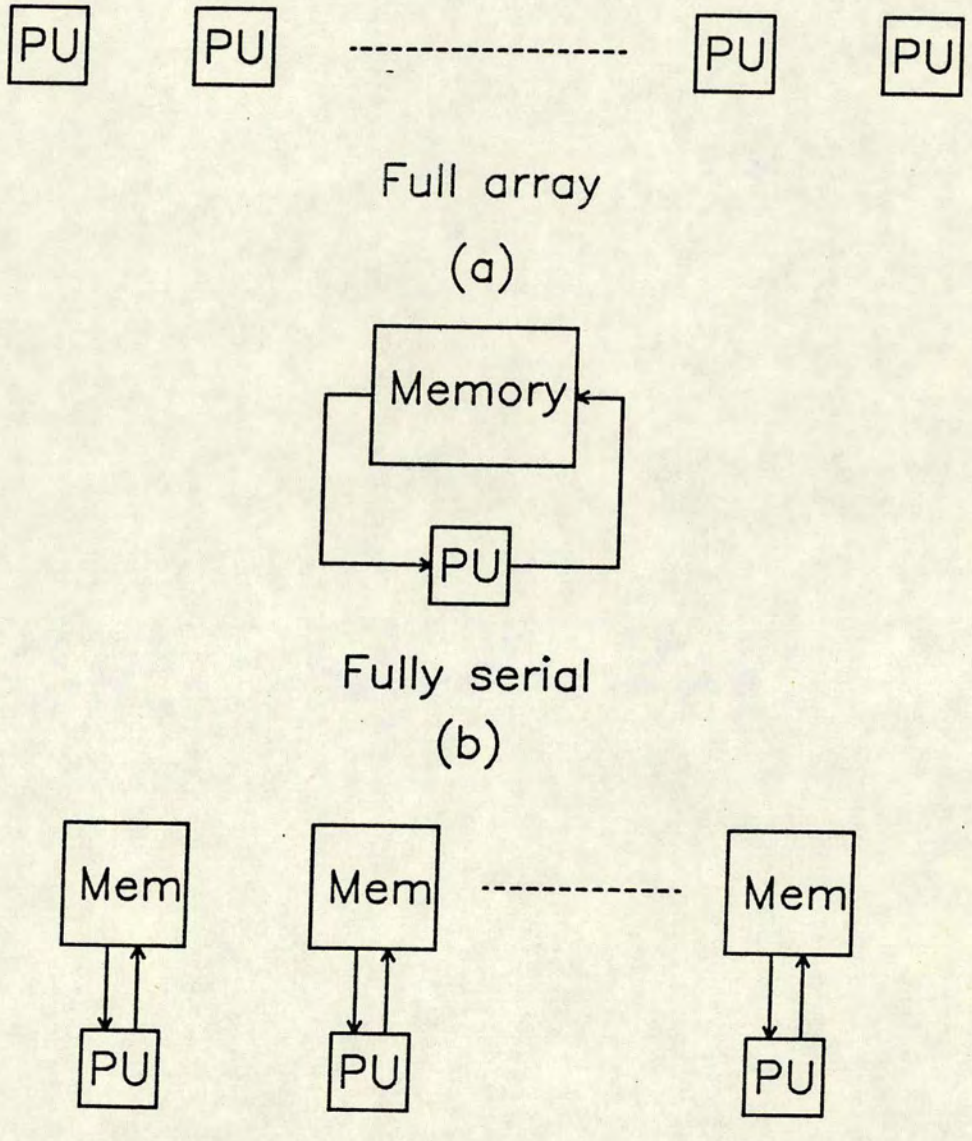

PU
PU
PU
PU
Full array
(a)
Memory
PU
Fully serial
(b)
Mem
Mem
Mem
PU
PU
PU
Multiplexed
(c)

Figure 4.1

one multiplier and one adder; this would be a fully serial architecture (see Figure 4.1(b)). Between these extremes there is a variety of choice, e.g., 2, or 4, or 8,... multipliers and adders could be used, which would give multiplexed architectures (see Figure 4.1(c)). Since the arithmetic elements have a fixed maximum operating speed, the choice of architecture limits the external bandwidth of the full function. Conversely, given a maximum sampling frequency for the filter, the choice of how much concurrency to use can then be made.

There are some further important considerations at this stage of design. If more than one processor is needed, the costs of inter-hardware communication must be considered; in other words, what balance is required between computation bandwidth and input-output bandwidth? What constraints are there on computational latency? Is there a requirement for extensibility and in what form; i.e. should there be a facility for adding more iterations by adding more hardware, requiring a modular solution, or is fixed order hardware required to cope with arbitrary order computation merely by alteration of the control structure?

For the purpose of this illustration assume that an appropriate solution is obtained by choosing to implement 32 iterations per processor. Thus 8 such processors are required to effect 256 iterations. Given this constraint, one possible computation scheme is then to have each processor evaluate successively 32 consecutive iterations of multiply accumulate: processors 1,2,3... evaluating iterations 1,33,65,... followed by 2,34,66... etc. These groups of iterations, carried out in one processor, are referred to as stages of the computation. The stages can be organised to be computed

concurrently, as here, or sequentially. The computation is then completed by adding the accumulation sums from the 8 stages. The details of this computation can be detailed in the form of a location-time-value graph, or hodograph (see section 4.4.1). The computation implicitly defines the connections required between processors as well as the memory requirements for each processor; thus, it defines the associated architecture. The hardware flow diagram of this architecture, shown in Figure 4.2, is derived by filling in these connections and memory elements explicitly. The memory required is particularly simple and can be implemented as FIFOs.

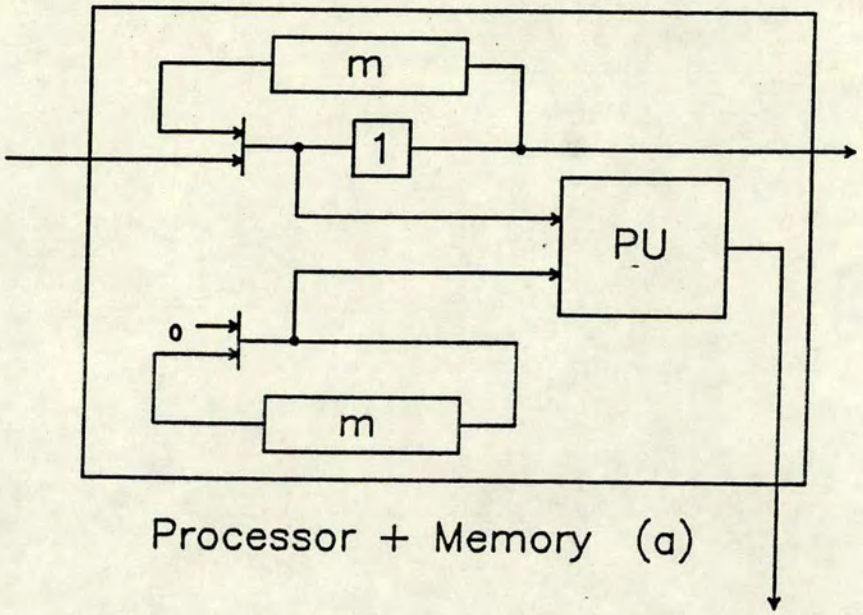
4.3. From Function to Architecture via Algorithm

4.3.1. Some Terminology

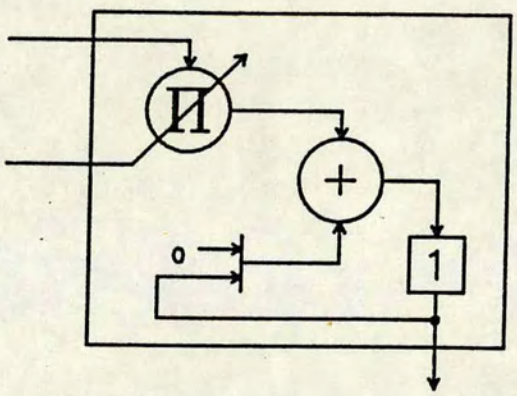
The following terms are used in a specific sense throughout this chapter.

The term function is used to mean a mathematical formulation of a signal processing transform. Such a formulation will not be interpreted as containing information about the way in which evaluation is to be carried out; it will be taken to define only the final result.

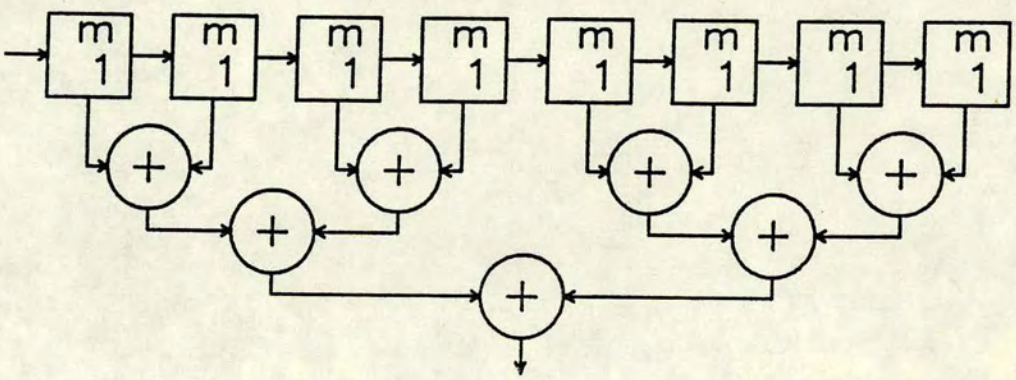
In contrast, the term algorithm is used to refer to the detailed, multiple sequences of arithmetic operations which are required to evaluate a function. An algorithm includes time, process location and control information and so determines the organisation of specific arithmetic operations. Algorithms can be expressed in a variety of ways including, parallel programming



Processor + Memory (a)



Processor (b)



Multiplexed architecture (c)

Figure 4.2

languages.

The term architecture is used to denote the organisation of specific hardware to carry out an algorithm, or class of algorithms. This organisation of hardware will have a direct correspondence with the organisation of concurrent computation. The main characteristics of an architecture are its "spatial" and "temporal" organisation. The former relates to the existence and interconnect of structural components, processors, memory, etc. The latter relates to their function in time and their control.

A machine is the actual hardware implementation of an architecture.

The class of functions for which architectures are discussed comprises those functions which can be formulated as recurrences.

4.3.2. Recurrences

A very broad class of signal processing functions can be expressed as a set of recurrence equations, with possibly also some initial and, or final, non-recurrent arithmetic processing. Such formulations are in wide use, as for example in time difference equation formulation of signal processing functions. The formulation of a function into a set of recurrence equations constitutes the first step of finding an algorithm for its computation. The heart of such a formulation is a basic set of arithmetic operations which are carried out many times for each evaluation of the function. Each repetition of this basic set of operations is termed an iteration. The total number of iterations required for one sample output of the function is referred to as the number of iterations.

In the example of section 4.2 there are 256 iterations of the multiply-add operation. The number of iterations is always fixed for the range of functions considered here. The operations carried out in one iteration define the processor function. In multiplexed architectures, where groups of several iterations are evaluated in a single processor the former are said to constitute a stage.

The sample cycle is defined by the duration of the arithmetic processing which must be carried out from the start of one input data set to the start of the next. Computation of iterations of the recurrence (together with any non-recurrence, peripheral processing if any is required) occur during this period. The computation done during one sample cycle may comprise the evaluation of all iterations for the current time sample, as in parallel or serial-parallel architectures. In this case the output data set will appear at the end of the current cycle, and the architectural latency is equal to the sample cycle, i.e., minimum. Alternatively the computation may comprise only one or a few iterations associated with the current time sample, as in pipelined architectures. In such cases the remaining iterations of the current time sample are evaluated previously and/or subsequently, using only part of the machine at a time. The data output set at the end of the current cycle may refer to the present or to some previous time sample.

Indefinite repetition of the cycle constitutes the machine function. Other cycles which will be referred to are function cycle, idles cycle, stage cycle, iteration cycle, word cycle and single full recurrence cycle. Processor latency is defined to be the time required for a processor to evaluate one iteration,

processor latency is equal to iteration period. Recurrence latency is defined to be the time required for the whole architecture to evaluate a single full recurrence, i.e., from present sample associated input to present sample associated output. The recurrence latency is equal to the full recurrence period.

Two specific recurrence formulations are possible: one in terms of a single time sample associated output of the function and the other in terms of the processor input-output "register" values. The former will be termed function recurrence equations and the latter processor recurrence equations. For function recurrence equations only time and iteration indices are needed; for processor recurrence equations time, processor and sub-cycle (multiplex, or idles) indices are needed (see the following subsections). Usually design proceeds by formulating function recurrence equations, choosing an architecture type and then formulating the corresponding processor recurrence equations. The equivalence of the two formulations can be established by induction on the processor recurrence equations. A hodograph (see section 4.4.1) gives a "visual" demonstration of the equivalence, and can be a useful development tool in formulating the processor recurrence equations.

An Index Notation for Iterations

Each iteration of a general recurrence requires inputs and produces outputs. In order to be able to refer to input and output values for any particular iteration the following notation is used. Let the total number of iterations be M, (in the introductory example M=256). Let i be the index denoting the iteration and let t be the index denoting the time sample association.

The iteration index i runs from 1 to M and will be interpreted as follows. The index i refers to the particular iteration under consideration. Any indices i-h refer to previous iterations and indices i+h refer to subsequent iterations.

The index t runs from 1 to any arbitrary value; the machine is assumed to cycle for an indefinite time. Sample time indexing is interpreted as follows. The index t refers to the present time sample, for which the full recurrence is being evaluated.

For the evaluation of the iterations of a full recurrence certain restrictions can be imposed on the inputs to each iteration. The criterion governing these is that of non-contradiction. For example, the value of the present iteration cannot be used before it has been calculated. Similarly, in a recurrence where the value of the previous-iteration-present-time-sample is required as an input for the calculation of the present-iteration-present-time-sample, the value of the next-iteration-present-time-sample cannot also be used as input. Suppose that the present-iteration-present-time-sample is indexed (i,t) then, under the assumption that all previous time sample iterations have been completed before the present one is commenced, outputs associated with the following iterations and time samples can be used in association without contradiction:

(i-h , t-s)

(i+h , t-s)

(i , t-s)

(i-h , t)

where h , s > 0 and 0 <= i <= M

i-h refers to previous iterations

i+h refers to subsequent iterations

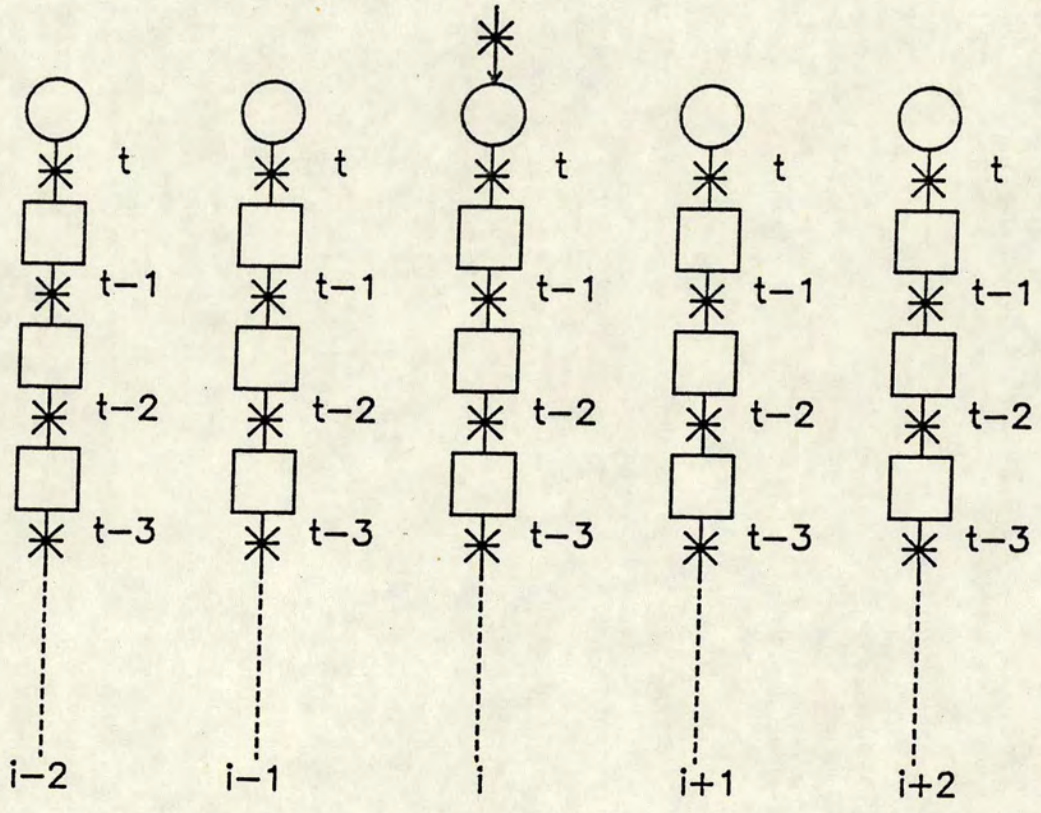
t-s refers to previous cycles

That is, the following outputs may be used: those from previous iterations at previous times, subsequent iterations at previous times, the present iteration at previous times and previous iterations at the present time. The iteration index i = 0 refers to external, initial value inputs, and M is the total number of recurrence iterations in one recurrence evaluation of the function.

Figure 4.3 shows representation of the present, previous and subsequent iterations with their time sample outputs and illustrates these valid connection points diagrammatically. A particular algorithm "programmes" the connections from these points to the inputs, using only those required to define it.

There are cases where it is necessary to relax the assumption that all previous time sample recurrences have been completed before the present one is commenced. In particular this is necessary in pipelined architectures. However, when this is done, restrictions will generally be imposed on the values that h and s may take.

Define the implementation latency to be the time taken to evaluate all iterations of the present sample time recurrence.



"uncommitted" processor & delay chain bank showing
iteration values available assuming

1) ascending order of evaluation

2) completeion of previous time sample associated iterations

Positions marked * are legal candidates for input to iteration I at time t.

Figure 4.3

Synchronisation of Recurrences with External I/O

In implementing recurrences, it is usually assumed that inputs which are external to iterations are supplied in the required synchronisation, and that the rest of the system is configured to ensure that this will happen. This must happen in real-time, so that firstly the computation bandwidth must be designed to be adequate and secondly any synchronising delay paths must be added. Often, particularly in adaptive systems, it is necessary to process the outputs from one cycle of a recurrence and feed back some resulting parameter(s) as input to the next cycle. The processing involved imposes a fixed delay between the end of one recurrence cycle and the start of the next. In order to ensure synchronisation of this output with the start of the next recurrence cycle there must be a phase of no operation for each processor during each cycle. In architectures which are continuously clocked, it is necessary to introduce extra, "dummy" iterations in the recurrence. These are carried out by the recurrence hardware during the imposed delay between the end of one recurrence cycle and the start of the next. For synchronisation reasons this delay must always be an integral number of iteration cycles. These will be referred to as idling iterations or idles, The recurrence cycle is then divided, with respect to the total number of iterations into a function sub-cycle and an idles sub-cycle which are contiguous and non-overlapping.

Clearly, idling iterations or no operation phases represent an unavoidable inefficiency in the implementation architecture. It is, therefore, necessary either to minimise the length of the idles cycle or to find a system use for the idles hardware during the

idles cycle, where these cannot be avoided altogether.

4.3.3. Recurrence Architectures

The form of a recurrence may determine what possibilities there are for organising the iterations and the processors which carry them out and whether they have to be carried out sequentially, or whether some or all might occur concurrently. Within the loose constraints imposed by the given function definition, there are usually several possible organisations for computation and architecture. These are now explored, so as to obtain a taxonomy of architectures.

In the introduction to this chapter architectures have already been classified as full array, multiplexed or fully serial, according to the number of processors used. This may be regarded as a kind of spatial classification. This is now extended further in relation to the distribution and sequencing of iterations within processors, i.e., in the direction of "temporal" classification.

Consider the iterations being evaluated during a single sample cycle in the processors. If these all relate to the same time sample output then the architecture will be termed homogeneous. Here all the iterations relating to the present output sample are carried out within the one recurrence cycle. If each iteration relates to a different time sample output the architecture will be termed separated. If some iterations refer to the same time sample output and some refer to different time sample outputs then the architecture will be termed grouped. Figure 4.4 illustrates this taxonomy of architectures. The above terms are chosen so as to avoid the ambiguity which more familiar designations hold.

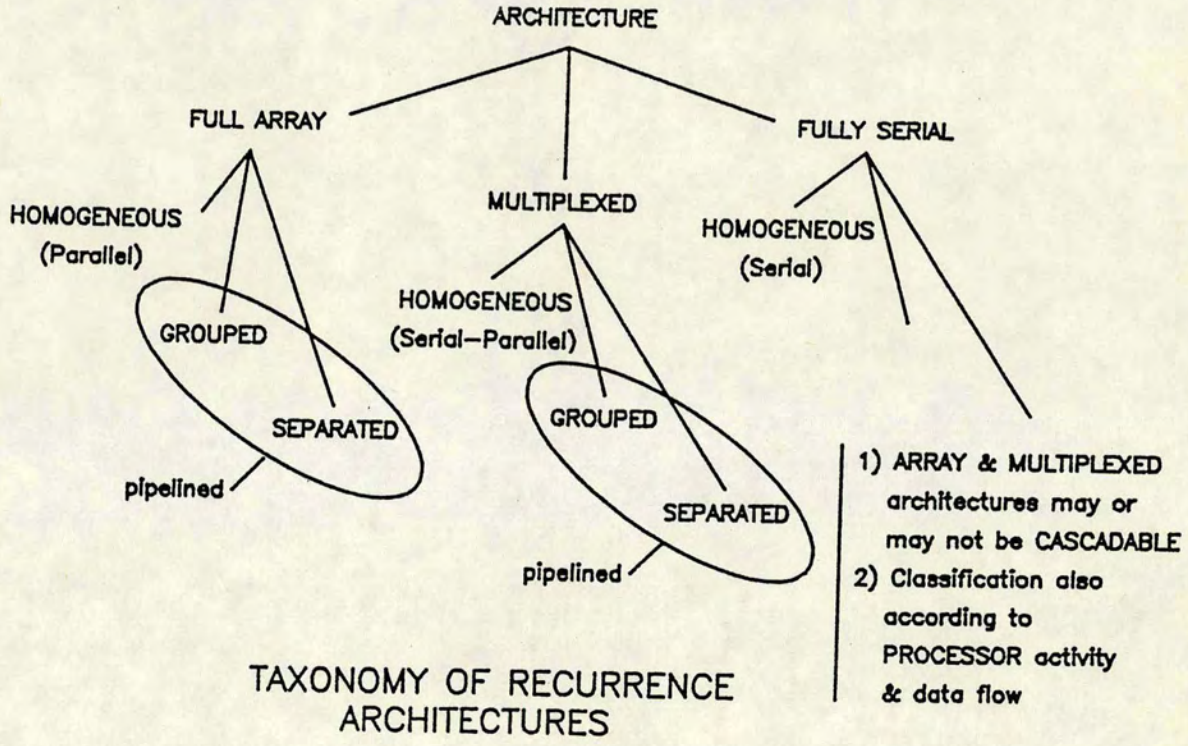

ARCHITECTURE
FULL ARRAY
MULTIPLEXED
FULLY SERIAL
HOMOGENEOUS
(Parallel)
HOMOGENEOUS
(Serial–Parallel)
HOMOGENEOUS
(Serial)
GROUPED
SEPARATED
pipelined
GROUPED
SEPARATED
pipelined
1) ARRAY & MULTIPLEXED architectures may or may not be CASCADABLE
2) Classification also according to PROCESSOR activity & data flow
TAXONOMY OF RECURRENCE ARCHITECTURES

Figure 4.4

However, they are somewhat cumbersome so the following conventions are adopted. Homogeneous full array architectures will be termed parallel architectures. Homogeneous multiplexed architectures will be termed serial-parallel architectures. Homogeneous fully serial architectures will be termed serial architectures. Grouped and separated architectures will be termed pipelined architectures.

The use of the terms parallel, serial-parallel, serial and pipelined should not be confused with their more general designations. For example, there is parallel processor function in homogeneous, separated and grouped architectures (if they are not fully serial) but the parallel function operates differently with respect to the time association of the iterations being evaluated. In one case it may be parallel and in another serial etc.

Let the number of processors in the architecture be denoted by the letter k, and let $1 <= k <= M$, where M is the total number of iterations. The architectures of particular interest are as follows.

Homogeneous fully serial: Serial

If k=1 there is just one processor and the present sample associated iterations must be computed serially in some order, usually either ascending or descending order. Ascending order sequential evaluation will be referred to as forward multiplexing of the processor to evaluate the present sample time associated iterations. Descending order sequential evaluation will be termed backward multiplexing. The implementation latency will be M processor cycles.

Homogeneous Multiplexed: Serial-Parallel

If 1<k<M there is more than one processor but not as many as one for each iteration of the recurrence. If, further, for the duration of the present time sample cycle the processors evaluate only present time sample associated iterations, then the organisation will be referred to as serial-parallel. Serial-parallel architectures can implement forward or backward multiplexing within each processor, all processors normally implementing the same type. The implementation latency will be M divided by k processor cycles. Serial-parallel architectures are instances of multiplexed architectures. In the introductory example k=8 and M=256

Homogeneous Full Array: Parallel

If k=M then there is one processor for each iteration of the recurrence, and for the duration of the present sample time cycle each processor evaluates only present sample time associated iterations, then the organisation will be termed concurrent or parallel. The implementation latency will be only one processor cycle.

Separated and Grouped: Pipelined

If, for the duration of the present sample cycle, the processors evaluate not only the present sample iterations, but also others, then the organisation will be termed pipelined. It is not usual to use a pipelined organisation with k=1. However, various pipelined organisations with k=M or with 1<k<M are possible. The implementation latency will depend on both the value of k and on the pipeline organisation. Often pipelining causes an increase in latency, but this may be an acceptable trade-off against increased

efficiency in other areas e.g., throughput, communication etc. Pipelined architectures may be multiplexed architectures or full array architectures.

In a multiplexed implementation, whether pipelined or serial-parallel, a single physical processor computes, sequentially, several iterations of the recursion and there are certain necessary restrictions on the values of k and M. The value k must be a factor of M. Clearly the choice of M as well as the choice of k has implications for multiplexed implementations. Often a signal processing algorithm, when expressed as a set of recursions, gives the specification of M in terms of an inequality, M being greater than some fixed value. Thus, the choice of value for M has a degree of latitude and a suitable value, which can be factorised, may usually be found to suit the implementation requirements. Often M is associated with function performance, so that it is possible to trade off performance against architectural efficiency and convenience.

Where an implementation uses more than one processor, it is usual for the multiple processors to work in synchronism. It should be noted that this is not the only possible organisation, other schemes include alternate processors active in antiphase over a two phase cycle, pipelined staggering of processor cycles etc. The specification of these details will be referred to as the pattern of processor activity.

It is worth noting here that the organisation of the sequence of iterations determines the type of memory needed to support each processor and that both the architecture and the recurrence deter-

mine the details of processor interconnect.

4.3.4. Recurrence Types

Recurrences are now classified according to their input requirements for the evaluation of iteration i at time t. This is important because the range of possible architectures is constrained according to this classification. Recurrences are classified as follows :

Class A: (i-h, t) is not used. That is, outputs from previous iterations in this time sample are not used for the computation of any of the iterations during it.

Class B: (i-h, t) is used but (M ,t-1) is not used as input to (j,t) for any value of j. This class exhibits zero sample time delay between iterations, but does not have recursive feedback from the last iterations.

Class C: (i-h, t) is used and (M ,t-1) is used as input to (j,t) directly or indirectly for some value of j. This class exhibits zero sample time delay between iterations and recursive feedback from the last iteration.

4.3.5. Bandwidth Matching

The key implementation issue has already been identified as that of concurrency, i.e., the issue of how many processors to use and how to organise them. Various types of architecture have now also been defined in relation to the number of processors used and in relation to the structuring of iterations. The issue of bandwidth matching is addressed next.

So that various architectural alternatives for implementing recurrence functions can be explored a notional, or virtual processor is associated with each iteration of the recurrence. By way of contrast, the final computational hardware for evaluating a single iteration of the recurrence will be referred to as a physical processor. In connecting up virtual processors no delay is associated with their operation. Physical processors, on the other hand have a fixed positive latency.

If it is decided to use one physical processor for each virtual processor, then the architecture must be either a parallel or a pipelined full array architecture. This usually results in excessive bandwidth for a given application. If this is the case then it is better to use fewer physical than virtual processors. The architecture is multiplexed and may be either serial-parallel or pipelined. Finally, if only one processor is used the architecture is serial.

Fixed rules can be developed for mapping full array architectures of parallel or pipelined type into multiplexed architectures. These rules are dealt with in detail in a later subsection; they allow easy and flexible adjustment of an architecture to cover a wide range of bandwidths. The notion of a group of virtual processors being replaced by a group of physical processors can help to visualise the mapping method and ease design detailing.

Recurrence Imposed Constraints on Bandwidth Matching

Firstly, when considering bandwidth matching in relation to choice of architecture, there are some weak conditions imposed by recurrence type. These restrict the architectures which can be

used to implement the various classes of recurrence and are as follows.

Class A recurrences can be implemented by any of the architectures.

Class B recurrences cannot be implemented by serial-parallel or parallel architectures and must therefore be implemented either in pipelined or serial form.

Class C recurrences can be implemented only serially.

Choice of Architecture & Multiplexing.

Once the recurrence and peripheral processors have been designed, their latency and the system word-length can be determined. This information is needed in order to determine how many processors to use. Meantime the key unknown quantities are represented by the use of symbolic constants. An outline is now given of how to match the required system bandwidth with the processor bandwidth in order to choose an architecture which has minimal hardware but meets the required performance. This usually turns out to be a multiplexed architecture, hence the association of the term multiplexing with bandwidth matching and choice of architecture.

The following symbolic constants are the key independent variables in a determination of the architecture.

Maximum system sampling frequency this is usually determined from the application bandwidth. Let B denote the maximum bandwidth required for the application and let F denote the minimum sampling frequency necessary to achieve this bandwidth. Then F is given by: F>=2B. The value F/2 is referred to as the Nyquist frequency. Often the sampling frequency is taken to be greater than this value.

Input signal quantisation, Q, measured in bits.

Hardware implementation maximum serial clock frequency, H, measured in Hz.

From these it is possible to determine the choice of implementation as follows.

Firstly, it is necessary to determine the processor latency, i.e., the processor bandwidth or cycle time. This is done by implementing the arithmetic processing required to calculate a single iteration of the recurrence and finding its latency. Let L denote this latency and GLB(L) denote the greatest lower bound on this latency, both measured in bits. Again, the eventual value of L must make up an integer number of iteration sub-cycles whereas GLB(L) need not. The value of GLB(L) will be determined by the minimum processing latency for implementation of the arithmetic function, whereas the value of L must take account of other issues.

Secondly it should be decided, from the inputs to the recurrence, whether it is necessary to have any idle iterations. Let z be the number of iterations of idles. The value of z will be determined by whatever arithmetic processing has to be done between

the completion of a full recurrence and the start of the next one and can be determined from the minimum latency of the hardware which implements this arithmetic processing. Denote this minimum latency by GLB(l), standing for the greatest lower bound on l, where l will be used to denote the latency of the hardware. The value of l in bits must always make up an integer number of sub-cycles, whereas GLB(l), also measured in bits, may not. The value of z must be an integral number of sub-cycles. Consequently, the hardware may have to be adjusted later, from having latency GLB(l), to having the next larger integral sub-cycle as latency, in order to satisfy this condition. This is easily done by the addition of FIFOs of the required length to each output. The value of z is given by:

$$z = GLB(l)//n \qquad \text{if } rem(GLB(l),n) = 0$$

$$z = GLB(l)//n + 1 \qquad \text{if } rem(GLB(l),n) \neq 0$$

where n is the system word-length, where GLB(l) is the greatest lower bound on the latency, where GLB(l)//n is the integer quotient and where rem(GLB(l),n) is the remainder.

Given the above groundwork, the following lemma expresses the number of processors to be used, in terms of the independent constants.

Let the number of iterations per stage be denoted by m, i.e., m is the number of iterations to be implemented by each physical processor. Let the total number of physical processors required to implement the M iterations of the whole recurrence be denoted by k. The following lemma states how to find values for m and k from the independent variables and expresses them in terms of the appropri-

ate symbolic constants. It should be noted that a description of the recurrence in these terms does not, in fact, constrain the eventual solution to being a multiplexed implementation; if m=1 and k=M then there is one processor per iteration and the implementation can be concurrent.

Lemma 1.

The solution sets for m and k are given by :

$$m + z \leqslant \frac{H}{nF} \qquad 5.3$$

$$k \geqslant \frac{M}{m} \qquad 5.4$$

Note that both m and k must be positive integers.

The above lemma states that the period required to calculate one iteration is n/H and the sampling period is 1/F , so that the number of iterations that can be calculated in one sample period is H/(nF). Its significance is that this fixes the upper limit on the total number of function iterations plus idle iterations that can be evaluated in this time. Thus if a total of M iterations are to be evaluated, where M is greater than m, then more than one processor has to be used and the number of processors required is given by k. This implies that the architecture must be multiplexed if m is greater than one, or a full array architecture if m equals one. Also it requires that the recurrence imposed constraints allow implementation by these types of architecture. If not then the recurrence cannot be implemented for these clock rates, word-lengths and bandwidths.

From the above lemma the solution set for m can be determined. Where this solution set is non-empty the value chosen for m should be the maximum allowable value, thus minimising hardware and maximising multiplexing within the required bandwidth.

Note 1.

$m = 1$	implies a parallel architecture
$m = M$	implies a serial architecture
$1 < m < M$	implies multiplexed architecture
$m < 1$	is NOT realisable

Note 2.

If the following conditions are satisfied, then the recurrence is real-time implementable as a pipelined array of multiplexed processing units.

1. $m >= 1$
2. all inputs for output indexed (i,s) have one of the following types of index:

   $(i - h, s - t)$

   $(i - h, s)$

   $(i + h, s - t)$

   $(i, s - t)$

   where $h, t > 0$
3. no conflict exists between implementation type as determined by the value of m

and as determined by the recurrence class.

Rules for Transforming Array Architectures into Multiplexed Architectures.

It has already been mentioned that fixed rules for mapping array architectures into multiplexed architectures can be developed. They are useful in taking a known high bandwidth array architecture and reducing both bandwidth and hardware in order to match specific lower bandwidth applications with minimum hardware. In this section a method for accomplishing this is outlined and some of the rules are stated.

Here attention is restricted to two types of full array architecture: the parallel architectures and the separated full array architectures. The full array is to be thought of as a virtual architecture and a method is demonstrated for distributing the operations of several processors of the full array into a single physical processor by time multiplexing and the addition of appropriate memory and control. The sequence of iteration evaluation determines the data flow beyond that already fixed by the full array architecture and thus fixes the memory structures required. It is of paramount importance, where there is freedom of choice with respect to this sequencing, to choose so as to minimise and simplify the memory requirements. When this is done, in many cases the memory can be implemented by nothing more complicated than FIFOs. For functions and array architectures where these simple memory structures cannot suffice either

special custom memory architectures can be developed or the conventional techniques using addressed RAM (random access memory). In such cases RAM address generation can become a substantial part of the design.

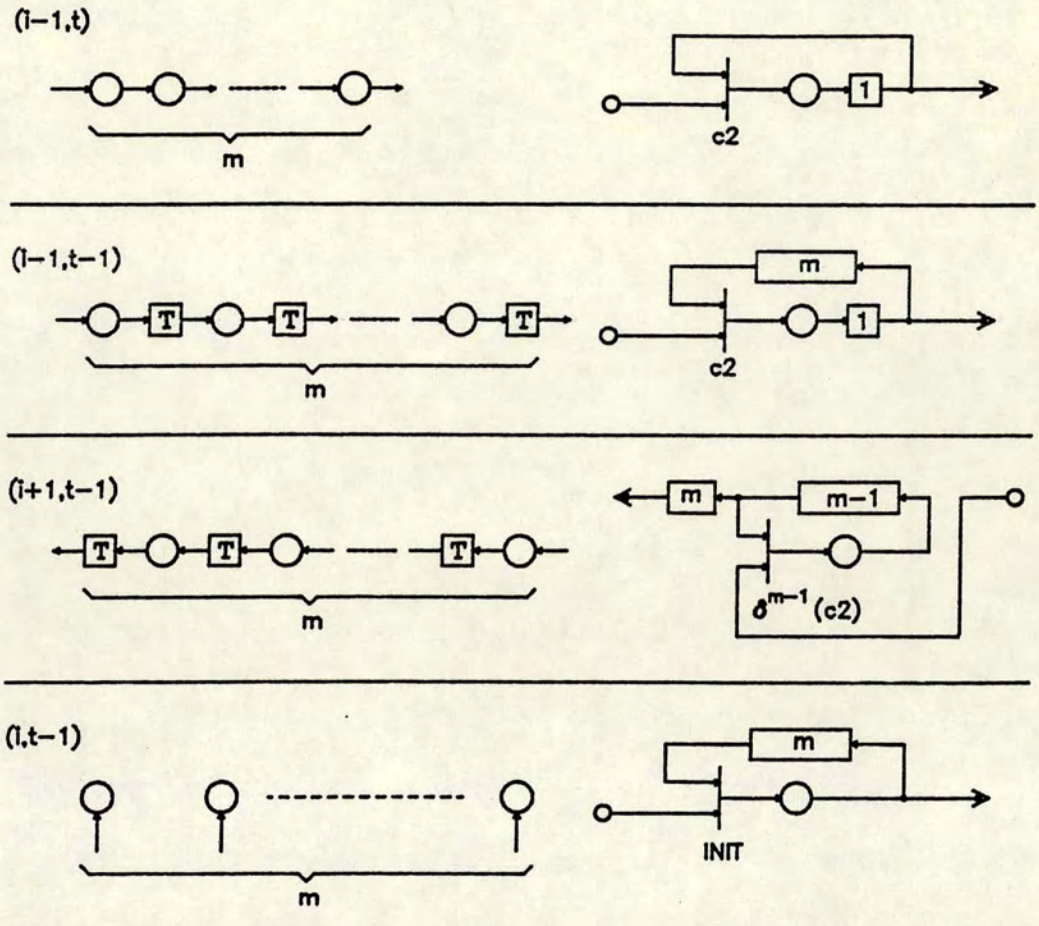
Consider again a recurrence of M iterations implemented as a full array architecture. Suppose that the equations given are the processor recurrence equations associated with this full array. Further, consider this as a virtual architecture for which a multiplexed architecture is sought able to carry out the same function. Let this multiplexed architecture consist of k physical processors each performing m iterations per sample cycle. The output of each iteration must be stored from the physical processor in such a way as to make it available later for input as required. This is fixed by the function and the virtual architecture. The following special cases occur frequently in the signal processing architectures of interest to us. They are listed together with their memory-multiplex-interconnect-control structure. The processor recurrence equations for the virtual architecture are indexed (i,t) by processor and sample, with $1 \leq i \leq M$. Since the array interconnection is regular, the processor recurrence input requirements can be formulated in terms of the processors indexed (i+h,t-s) or (i-h,t-s) whose outputs must be made available to the processor (i,t). Each of the three instances stated below is a special case of this with h=1 and s=0 or s=1. In particular, it should be noted that it is assumed for these cases that all processors in the multiplexed architecture are active for the whole of

the sample cycle. Generalisation to other values of h and s is straightforward. The hardware and control required for other cases, not covered by this list, for example with other patterns of processor activity, can usually be derived easily from a hodograph of the full array architecture.

In full array architectures requiring the value of iteration (i-1,t) as input for the calculation of iteration (i,t) m virtual processors are replaced by a single physical multiplexed processor with output fed back to input via a memory loop of length 1 iteration cycle and multiplexer controlled by c2. The cascade tap out point is as shown and if such multiplexed processors are cascaded then they will each be operating on groups of iterations associated with successive time sample outputs, see Figure 4.5.

In full array architectures requiring the value of iteration (i-1,t-1) as input for the calculation of iteration (i,t) m virtual processors are replaced by a single physical multiplexed processor with output fed back to input via a memory loop of length m+1 multiplex subcycles and multiplexer controlled by c2. The cascade tap out point is as shown and if such multiplexed processors are cascaded directly then they can all operate on iterations associated with the same time sample outputs, see Figure 4.6.

In full array architectures requiring the value of iteration (i+1,t-1) as input for the calculation of iteration (i,t) m virtual processors are replaced by a single physical multiplexed processor with output fed back to input via a memory loop of length m-1 multiplex subcycles and multiplexer con-



Note:

1) δ^m is m bit delayed version of c2
2) T→○ & ○→T are equivalent
3) T represents sample cycle delay
4) 1 & m represent 1 & m multiplex subcycle delay etc.
5) x0, x1 → y; y = x0 for c2 = 0, y = x1 for c2 = 1; c2 is control
6) → cascade o/p ○— cascade i/p

Figures 4.5 - 4.8

trolled by c2. The cascade tap out point is as shown and if such multiplexed processors are cascaded directly then they can all operate on iterations associated with the same time sample outputs, see Figure 4.7.

In full array architectures requiring the value of iteration (i,t-1) as input for the calculation of iteration (i,t) m virtual processors are replaced by a single physical multiplexed processor with output fed back to input via a memory loop of length m multiplex subcycles and multiplexer controlled by c2. The cascade tap out point is as shown and if such multiplexed processors are cascaded directly then they can all operate on iterations associated with the same time sample outputs. This type of loop is to provide constant coefficients associated with each iteration, see Figure 4.8. Note that, in the full array architecture, the pattern of activity for each processor can include only one active phase and m-1 idles phases during a single sample cycle

These special cases are now extended as follows. Represent the multiplexed architecture by a notional bank of "uncommitted" processor-memory chains as partially illustrated in Figure 4.9. The following lemma states where to find the correct tap out positions for input to processor (i,t) in the multiplexed architecture.

Lemma 2.

Consider a full array architecture consisting of M processors. Let processor recurrence equations for any output associated with the processor (i,t) be formulated in terms of

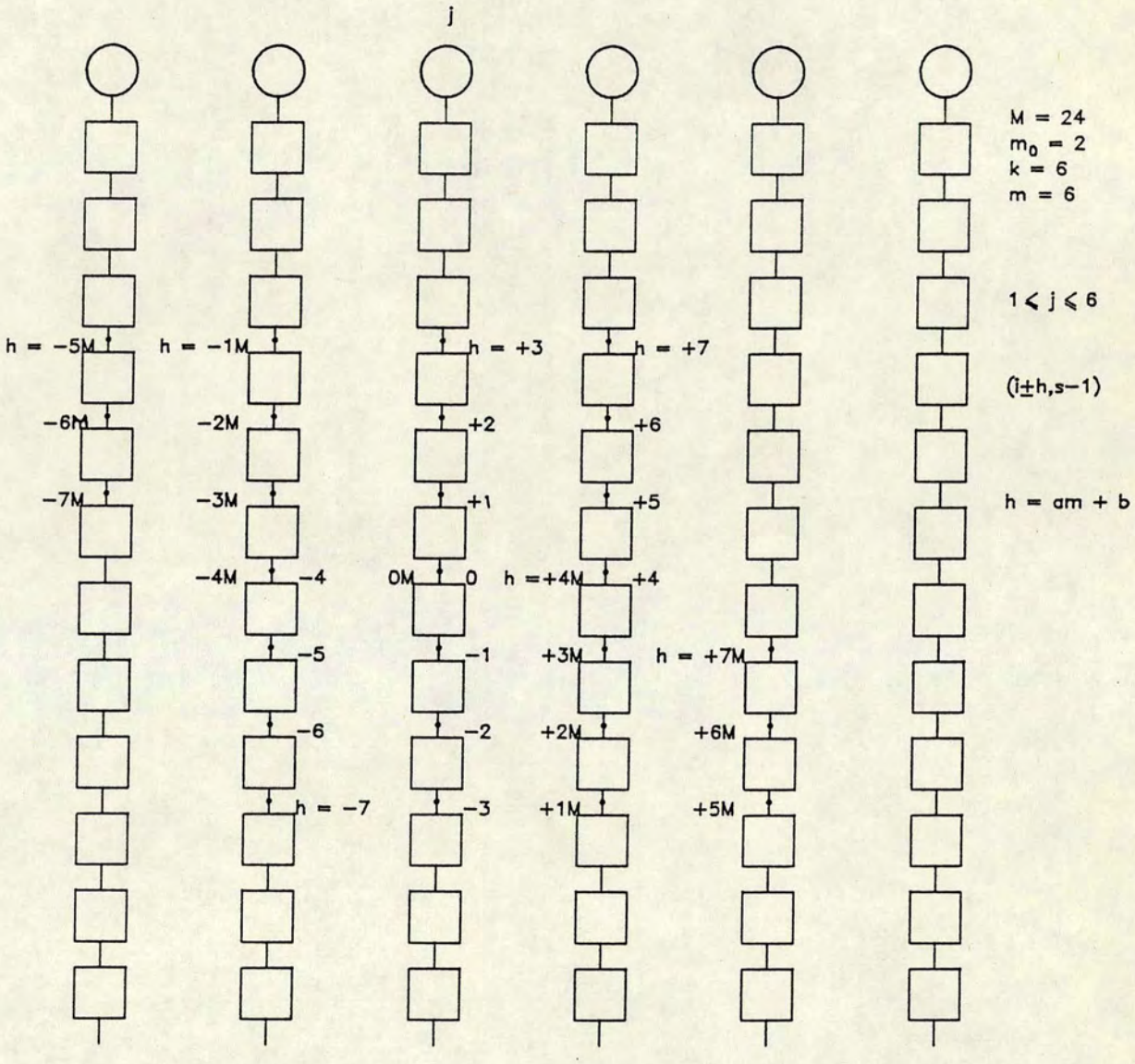

j
M = 24
m_0 = 2
k = 6
m = 6
1 ≤ j ≤ 6
(i±h,s−1)
h = am + b
h = −5M
−6M
−7M
h = −1M
−2M
−3M
−4M
−4
−5
−6
h = −7
0M
0
−1
−2
−3
h = +3
+2
+1
h = +4M
+4
+3M
+2M
+1M
h = +7
+6
+5
h = +7M
+6M
+5M

Figure 4.9

the outputs associated with the processors

(i-h,s-r)

(i+h,s-r)

(i , s-r)

(i-h, s)

Description	Input	
	Main (C2=0)	Alternate (C2=1)
(i-h, s-r)		
I/p Tap Position	j-a	j-(a+1)
Distance from MPX point	$r(m_0+m)+b$	$r(m_0+m)-m+b$
Selected during	$b < t \leqslant m_0+m$	$0 < t \leqslant b$
(i+h, s-r)		
I/P Tap Position	j+a	j+a+1
Distance from MPX point	$r(m_0+m)-b$	$r(m_0+m)+m-b$
Selected during	$0 < t \leqslant m-b$	$m-b < t \leqslant m_0+m$
(i, s-r)		
I/P Tap Position	j	not used
Distance from MPX point	$r(m_0+m)$	not used
Selected during	$0 < t \leqslant m_0+m$	not used

Table 4.1

Further, let the processor activity pattern be such that there is one function subcycle followed by m_0 idle subcycles per single sample cycle (i.e., the processors are active together and idle together). Then this full array architecture can be replaced by a multiplexed architecture consisting of k processors each implementing m iterations, where M=km. The single sample cycle length for the multiplexed architec-

ture will be (m_0 + m) and the processor activity pattern will all be m function subcycles followed by m_0 idle subcycles. Further, if the j-th processor of the multiplexed architecture evaluates the function of the (j-1).m + 1,...,j.m processors of the full array architecture in ascending order then Table 4.1 gives the tap positions of the notional multiplexed architecture according to input type. Let h = a.m + b where 0<=b<m and let t be an integer such that $1 \leq t \leq m_0 + m$. The Table 4.1 gives the input tap positions in terms of the processor to be tapped and the distance of the tap point from the processor multiplex point. It also gives the parts of the multiplex subcycle during which the selection is valid. Figure 4.9 illustrates this for m=4, $m_0 = 2$.

4.4. Miscellaneous Related Topics

In the foregoing sections an outline of a class of functions and architectures has been given together with methods for matching a given function to an efficient known architecture. This section covers two miscellaneous related topics. Firstly a technique is introduced which can be used for the verification of architectures and for developing new architectures. Secondly the main evaluation criteria for assessing signal processing architectures are set out.

4.4.1. Processor-Time-Value Graphs, Hodographs & Hardware Signal Flow Graphs.

The above approach to modelling computation naturally leads to a location-time-value indexed notation, which

expresses signal values as a function of the following arguments, or indices :

processor index

sample time index

multiplex sub-cycle index

Generally, signal processing equations can be reformulated, formally in terms of such notation, as processor recurrence equations. These recurrence equations imply corresponding hardware processors, memory requirements and interconnection of elements; if developed correctly they could be used as a high level form of hardware description. The time sample index is integer and arbitrary. Negative values will be used only to represent initial values, of internal state, etc. The processor index is integer and bounded between one and the total number of physical processors used in the implementation. The multiplex sub-cycle index values are integer modulo the multiplex sub-cycle length, and therefore lie between zero and one less than the sub-cycle length. The iteration index of the function recurrence equations is integer and bounded between one and the total number of iterations of the recurrence. The iteration index is directly related to the processor and multiplex subcycle indices through the architecture. In concurrent architectures, the iteration and processor index are identical, the multiplex cycle reduces to one iteration cycle, and the fundamental cycle is often also the same as these. Such notation allows the detailed function of the computation scheme to be represented and defined. In particular, the

simultaneous or non simultaneous staggering of processor operation, the distribution of intermediate values between processors, the order of computation of partial results within processors and the data flow communication between processors can be defined unambiguously.

One of the problems associated with such notation is that it involves a profusion of index detail and therefore tends to be tedious and error prone. Further, different processor recurrence equations often evaluate the same function and that function may not be obvious from the recurrence equations. Thus it is possible to have two or more sets of different recurrence equations, with corresponding different hardware implementations, to compute the same function and this fact will not be clear from the recurrence equation notation alone. Various techniques exist for making this processor-time-value notation easier to use. The representation of computation schemes can be expanded into tabular or graphical form by enumerating the processor index horizontally, (conventionally left to right ascending), and the sub-cycle index vertically, (conventionally top to bottom ascending). The cycle index is constant over each full sub-cycle count and then increments at the start of the next. Each sub-cycle count is evaluated modulo the sub-cycle length and therefore runs repeatedly from zero to one less than the sub-cycle length. The elementary unit of time is that of one processor operation. Within this framework the computation scheme is defined by entering the signal values computed at each processor-time position. This method of representation

will be termed a processor-time-value graph of computation. Automation of the processor-time-value graph notation can be used in a limited way to represent and manage values notationally by the use of a "symbolic" simulator.

A further useful simplification is obtained by suppressing the details of index notation and function value and to represent only the flow of data on the processor-time graph grid. This can be done by the use of arrows to show data transfer in time and between processors. Such graphs are particularly useful for concurrent or pipelined, laminar flow, nearest neighbour, regular interconnection schemes, but can become unreadable for non local, or multiplexing induced turbulent flow communication schemes. In this form of representation vertical arrows represent data held within a processor but passed forward through time, and diagonal arrows represent data which is being passed from processor to processor through time. This form of representation emphasises the interprocessor communication structure but also shows how it behaves through time. Such graphs are termed hodographs. On a hodograph the feature of particular interest is where lines of flow intersect, as this is where the corresponding data will interact within processors. Diverging or parallel lines of flow represent data items which cannot interact to form partial results. The hodograph is a useful aid to visualising the interaction of data flowing at different rates and in different directions through a linear array of processors.

Hardware implementation of the computation scheme can be

derived easily from any of the above representations by the addition of the required memory storage and correct interconnection (including multiplex-switches) between the processors. In its initial form this is conveniently represented as a form of hardware signal flow graph or interconnection net of processor, memory and multiplex-switch elements. There is a choice in representation as to whether to interpret and therefore represent processors as including latency or alternatively to interpret their function as instantaneous and represent the latency distinctly on each output. Each method has its advantages and disadvantages. The function of the memory and interconnect is to supply the correct input values to each processor at the correct time and is in essence defined implicitly by the computation scheme. Methods whereby this may be implemented are discussed below. Finally such hardware flow graphs can be coded directly into FIRST code for compilation.

The representation methods outlined above furnish useful notational tools for developing or synthesising concurrent computing schemes, for defining them and also for verifying their function. Later sections of this chapter will indicate details of the derivation of hardware specifications from such computing schemes. The next sub-section outlines criteria for evaluating the merits of any particular computing scheme and the following sub-section illustrates the use of what has been discussed by means of several examples.

4.4.2. Evaluation Criteria.

Evaluation of any computation scheme in relation to the proposed application is an important part of the design task. Such evaluation should be completed before final commitment to hardware and can be undertaken either before coding a hardware flow graph to implement the design or else on completion. The former is done if there is a well known theoretical approach for resolving all the issues, the latter if it is required to use a simulator in order to carry out some of this evaluation. Evaluation should include the following aspects and measure the associated attributes of the proposed architecture.

Hardware Evaluation.

total hardware required in each processor:
number, type and use of primitives
total local storage associated with processor
processor latency
processor modularity
total number of processors

Evaluation of Communication.

switching and interconnect features: extent,
locality, bandwidth
global communication overheads

Computation Evaluation.

data flow, input/output, communication

final output extraction

response time

throughput rate

duty cycle indication of use of processors

local storage

modularity

expandability: via addition of hardware, via reconfiguration

Evaluation of arithmetic performance.

system word-length

use of dynamic range, signal distribution

introduced non-linearities: overflow performance

noise performance

scaling overheads, format matching

stability

The examples in the next section illustrate some of the evaluation points with regard to computation schemes.

4.5. Examples.

A specific example is now taken in order to illustrate the application of the ideas described above. The function chosen is that of the introductory section. This example is used for two reasons, firstly it is a simple function so that extraneous complexities will not obscure the principles involved and secondly it can be implemented by all the different types of architecture, giving a unified illustration for making comparisons. The various architectures that are

set out below could be implemented in a variety of ways: bit-serial or bit-parallel, using discrete, programmable or custom parts. Some aspects of the evaluation of the computation architecture will depend partly on the implementation chosen. In particular, architectures with smallest input/output will be strongly favoured for bit-parallel systems whereas this may not be important for bit-serial systems, giving the designer an extra degree of freedom in finding hardware efficiency.

For the purposes of the following examples small, unrealistic values are chosen for k, m, M . This allows the diagrams to be more easily followed. Each allows arbitrary extension and the form of the processor recurrence equation representation is general.

The function, for which an architecture is to be found, is defined as follows. Given constants

$$w_1, w_2, \cdots, w_M$$

and given signal values

$$x_1, x_2, \cdots, x_N$$

where $N >> M$

define

$$y_i = \sum_{j=1}^{M} w_j x_{i+j-1} \qquad \text{for} \quad 1 \leq i \leq N+1-M \qquad 5.5$$

Firstly observe, by comparing the data and partial results required for the computation of successive values y_i that the function can be expressed as a class A recurrence. In the interests of illustrating a wide range of architectures bandwidth matching is not considered here. Instead the full array architectures are explored and then the multiplexed architectures are considered. The fully serial architecture will not be discussed. The computation of a single sample output y_i requires the evaluation and summation of M products $w_j x_{i+j-1}$. Thus in any architecture for this algorithm, these are the iteration operations and the iteration cycle is defined as the time required for the hardware to perform a single multiply-add. Answers to the following questions will then determine the architecture type and specific structure.

How many processors are required to match the sampling frequency with the individual processor bandwidth?

How are the iteration products and partial sums to be formed with respect to a single sample cycle?

How are the data values to be located in the processor array?

What pattern of processor activity is to be used?

Answers to these then determine the spatial and temporal aspects of the architecture fixing the data flow (i.e., they determine where data is used or formed during one cycle and where it must be sent for the next). From these requirements hardware can be derived directly. Apart from bandwidth restrictions, the range of possible architectures can be viewed in terms of the different ways in which the data can flow in order to compute the correct function, in this example the flow of x_i, w_i, Y_i to form y_i. The processors that particular data has to flow through determines interconnections and the rate at which data flows through them determines their latencies. A detailed representation can be worked out on a location-time-value graph or on a hodograph, a formal summary can be specified in terms of the processor recurrence equations and the corresponding hardware block diagram can also be given.

Within an array architecture a range of alternatives can be built up according to the range of possible flows of the data involved and the pattern of processor activity required

to compute the function using these flows. Often some specific flow patterns will exclude computation of the required function, or give rise to inefficient results.

A Note on Flow and Processor Activity

The flow of data with respect to a linear array of processors can be forward, backward or stationary. Constant rates of flow can be represented by positive, negative or zero values and measured in terms of the processor associated latencies as $1/n$ where n is the latency of each processor. Globally communicated data, corresponding to zero latency associated with processors, can be thought of as having infinite flow rate for notational purposes.

The sample cycle processor activity must be defined in terms of the total number of iteration and idle cycles performed between input samples. Processor activity can then be defined in terms of this cycle as the pattern of activity. A numerical value for this can be derived for each processor by coding the active and idle phases within the cycle. Active phases can be coded 1 and idle phases 0. For example, for a cycle consisting of five iteration and three idle cycles, a processor which is active during sub-cycles 1 and 3 would have code 00000101 . For maximum efficiency processors should be active throughout the sample cycle, but some architectures rely on alternate processors being active during different phases in order for data to flow correctly to form the required function. Example 4 below illustrates an instance of this.

The hardware flow diagrams for each example are given with the text in Figures 4.10 - 4.15.

Example 1: Homogeneous full array.

Equation 5.5 can be computed directly by the use of M parallel data inputs to M multipliers and by using a parallel M input adder to sum the products. This is, however, unsuitable for anything other than very small values of M, because of the hardware problems involved in parallel addition and parallel data input. A variant of this architecture is to pipeline the data input and implement it via a tapped delay line. The hardware for this architecture is shown in Figure 4.10(a) where Figure 4.10(b) shows the function of the repeated processors.

Example 2: Pipelined full array.

The main problem of the parallel architecture for the function given in equation 5.5 is that of the fan-in associated with parallel addition. The way to overcome this is to pipeline the formation of the sums by formulating each as "the sum so far" plus the next term and then to form it sequentially. One way of doing this is illustrated in Figure 4.11.

Examples 3 & 4: Pipelined full arrays.

Example 2 exhibits pipelining of the sum formation but still has the problem of fan out, for moderate or large values of M. This can be solved by pipelining both the sum formation and the data transmission and formulating them as

recurrences. Several different forms of pipelining and processor activity are possible and the range of these can be explored by a systematic investigation of the various flow patterns that can be used. Two cases are illustrated. Example 3, in Figures 4.12 illustrates one such design in which the flow rates are all non negative and the processor duty cycle is 100%. Example 4, in Figures 4.13 illustrate a radically different design in which one flow is positive one negative and the third zero, and the processor duty cycle has to be 50% for correct evaluation of equation 5.5. Other variants of pipelined full array architectures can be developed along these lines, starting from the data flow.

Example 5: Pipelined full array.

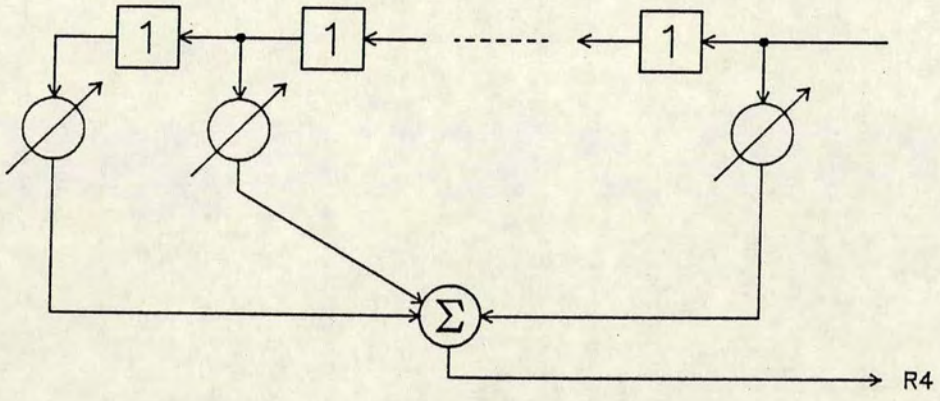
One feature of the architectures illustrated so far is that in order to increase M additional hardware must be added. At best this is just the connecting up of identical hardware as in the case of modular architectures, at worst it involves component redesign. For some applications it may be possible to relax the requirements on sampling frequency in the interest of extending M without having to add hardware. In other words a degree of programmability by control is wanted, rather than programmability by hardware reconfiguration. A simple example of such programmability by control can be seen in the comparison of extending a bit-parallel adder and a bit-serial adder to accommodate a longer word-length. Example 5, illustrates a pipelined architecture, with the same number of processors as iterations but which can be reconfigured to operate with fewer processors than

iterations. The pipelining is arranged so that each processor forms a fixed time sample recurrence and during any one cycle the processor form batches of successive time sample outputs. The weights and data samples are passed systolically from processor to processor at different rates; the partial results stay fixed and are accumulated in each processor. Successive processors output, in succession, batches of successive time sample outputs. This is illustrated in Figures 4.14.

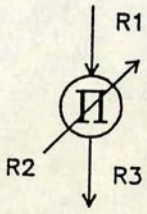
Example 6: Multiplexed pipelined.

So far all the architectures have been full array architectures. The remaining example, see Figure 4.15, shows a multiplexed architecture, suitable when the maximum bandwidth offered by the previous implementations is excessive and it is more important to minimise the amount of hardware. It also illustrates the technique for mapping a full array architecture into a multiplexed architecture, being a multiplexed version of the full array architecture given in example 4.

The examples given above by no means exhaust the range of possible architectures for evaluating the convolution sum of products but they do illustrate many of the many alternatives and their architectural features, merits and problems. Further examples of this may be found in [15].

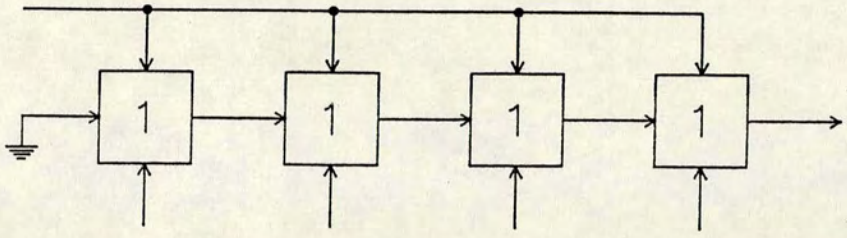


(a)

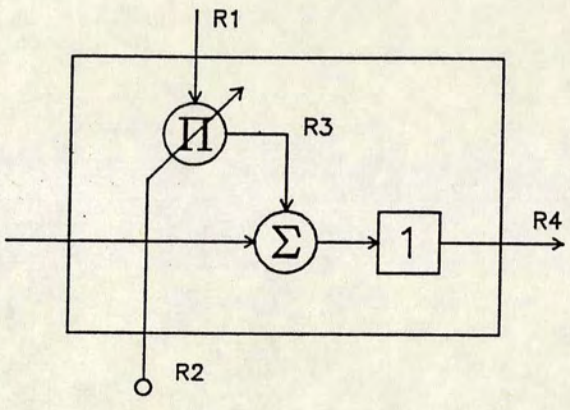


(b)

Figure 4.10

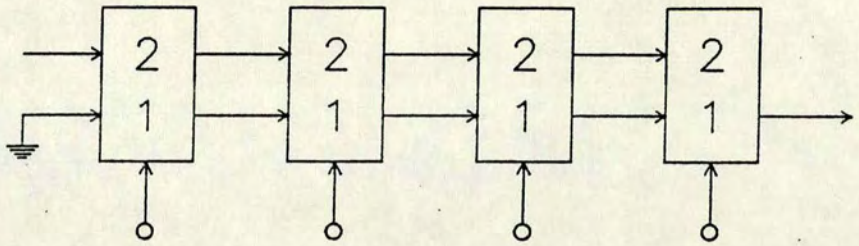


(a)

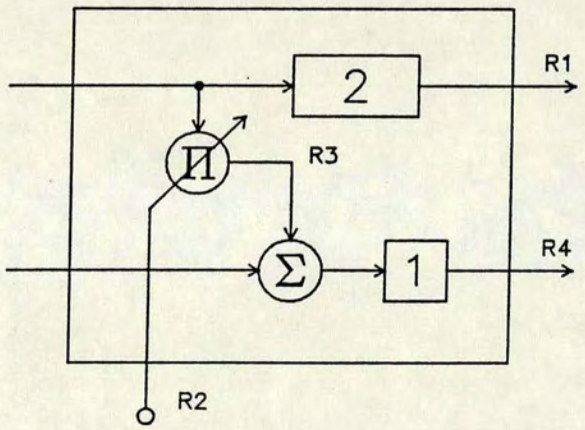


(b)

Figure 4.1

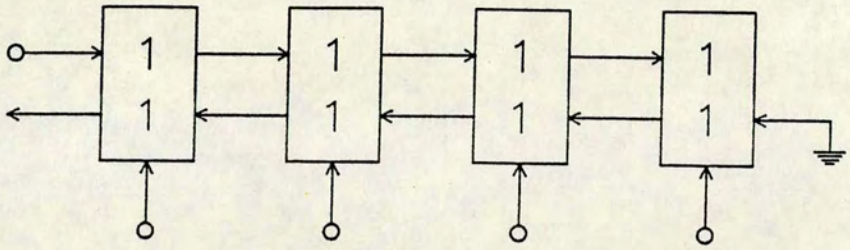


(a)

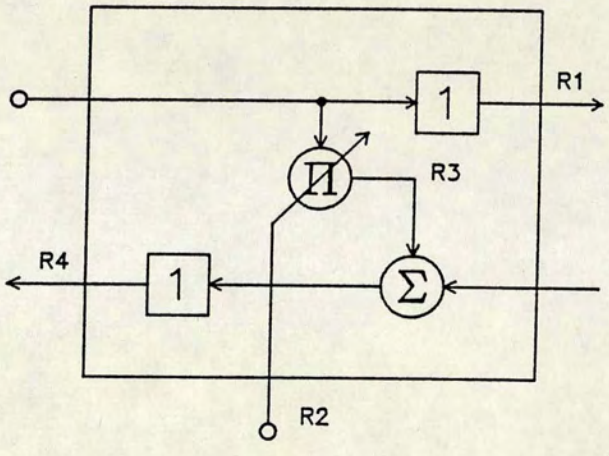


(b)

Figure 4.12

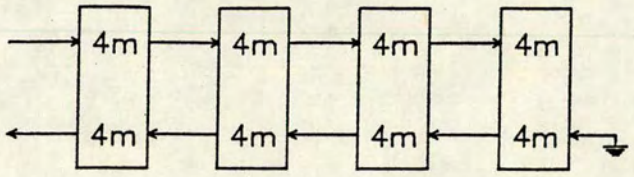


(a)



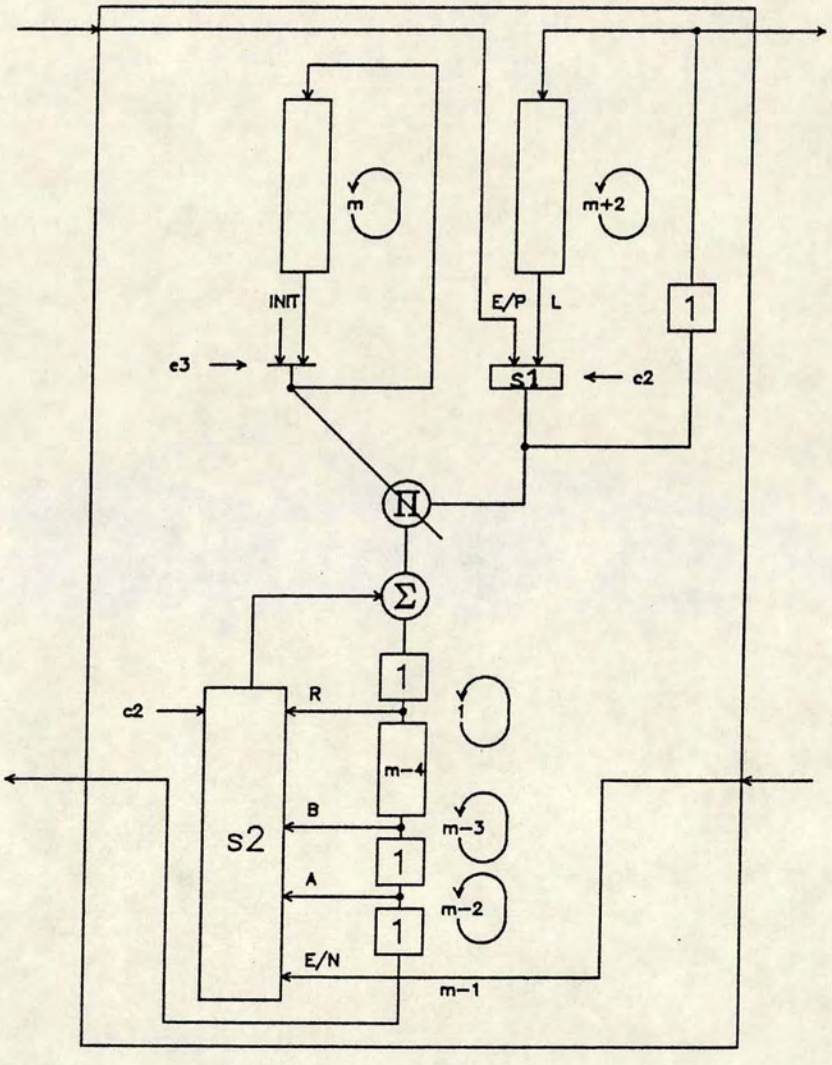
(b)

Figure 4.13



multiplexed array

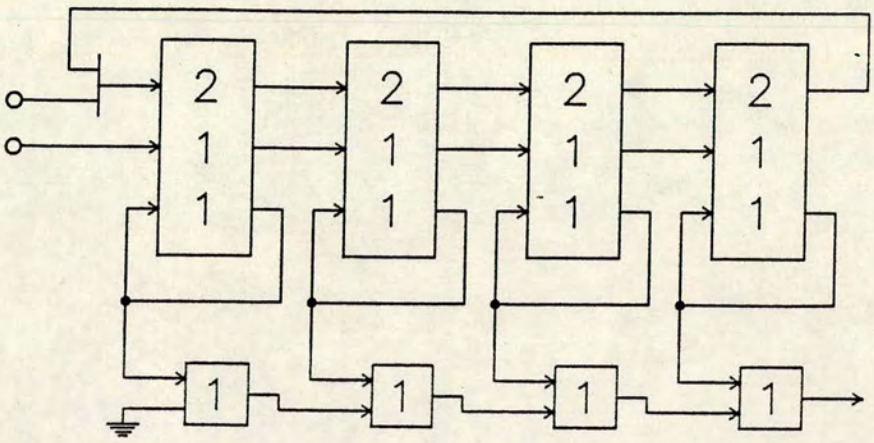
(a)



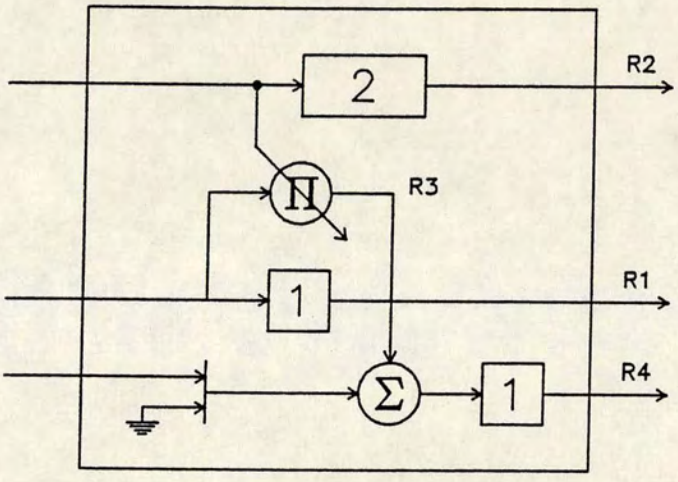
multiplexed processor

(b)

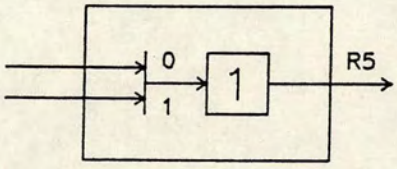
Figure 4.14



(a)



(b)



(c)

Figure 4.15

References

1. H. T. Kung, "Let's Design Algorithms for VLSI Systems," Department of Computer Science Report, Carnegie Mellon University (1979).

2. H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," pp. 271 - 292 in C. A. Mead & L. Conway, "Introduction to VLSI Systems", Addison-Wesley (1980).

3. D. Cohen, "Mathematical Approach to Computational Networks," Technical Report, University of Southern California, Information Sciences Institute (November 1978).

4. L. Johnsson and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," pp. 213-68 in VLSI Systems and Computations, ed. H. T. Kung, B. Sproull, G. Steele,Springer Verlag (1981).

5. L. Johnsson, U. Weiser, D. Cohen, and A. L. Davis, "Towards a Formal Treatment of VLSI Arrays," Technical Report, California Institute of Technology, Computer Science Department (January 1981).

6. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," pp. 131-140 in VLSI 81: Very Large Scale Integration, ed. J. P. Gray,Academic Press (1981).

7. P. B. Denyer and D. Renshaw, VLSI Signal Processing: A Bit-Serial Approach, Addison-Wesley (to be published).

8. R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Linear-Time GCD Computation," Proc. VLSI'83, pp. 145 - 155 (Trondheim, August 1983).

9. K. Bromley and et al., "Systolic Array Processor Developments," pp. 273 - 284 in VLSI Systems and Computations, ed. H. T. Kung, R. F. Sproull & G. Steele Jr.,Springer-Verlag (1981).

10. P. R. Cappello and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," pp. 245 - 254 in VLSI Systems and Computations, ed. H. T. Kung, R. F. Sproull & G. Steele Jr.,Springer-Verlag (1981).

11. A. L. Fisher and H. T. Kung, "Synchronising Large Systolic Arrays," SPIE Real-Time Signal Processing V Vol. 341 pp. 44 - 52 (1982).

12. W. M. Gentleman and H. T. Kung, "Matrix Triangularisation by Systolic Arrays," SPIE Real-Time Signal Processing IV Vol. 298 pp. 19 - 26 (1981).

13. R. H. Kuhn, "Yield Enhancement by Fault Tolerant Systolic Arrays," pp. 145 - 152 in Proc. USC Workshop on VLSI Design and Modern Signal Processing, ed. S. Y. Kung, (Los Angeles, 1982).

14. H. T. Kung, "Why Systolic Architectures?," Computer Vol. 15(1) pp. 37 - 46 (January 1982).

15. H. T. Kung, , Lecture Notes for Advanced Course on VLSI Architecture (Bristol, July 1982).

CHAPTER 5

SYSTEM SYNTHESIS AND SYSTEM MAPPING TECHNIQUES USING FIRST

5.1. Introduction

This chapter describes the design steps necessary for translating a computational system architecture into a functioning FIRST HDL specification. The form of the procedure given was developed by the author as a result of system design and consultation with P. B. Denyer and S. G. Smith. This chapter appears in a revised form in [1].

The method through out has been to create well defined interfaces at strategic boundaries in the system design hierarchy. Transformation of data from one interface to the next can only be automated if there is a set of formal rules defining the transformation. Properties which are preserved from one interface to the next must be invariant under the transformation and the transformation must only generate valid new data. The ease with which the problems between interfaces can be solved depends on the judicious definition of the interfaces. Where this has been done successfully the interface definitions are adopted as a standard, as for example CIF, RS232, file transfer protocol, etc. The main virtue of such an approach is to give a structured method for managing system complexity.

In the design hierarchy proposed here such interfaces are the function, the algorithm and its architecture, the flow-graph, the FIRST hardware description language and CIF. In Appendix III an example

is given, showing how to transform a signal flow graph into the FIRST hardware description language, and the silicon compiler then transforms this into CIF and an associated functional behaviour. In the chapter 4 some target system architectures have been developed; these assist in transforming the function into an algorithm and architecture. Having established the architectural context and structure for system implementation all that is left is the need to complete the details of the flow-graph. In this chapter a synthesis methodology is derived to provide a guided path through from computation architecture to hardware flow graph.

Overview

The plan is to build systems by :

Identifying the target architecture;

Building arithmetic engines (processors);

Supporting the processors with state memory;

Implementing the array of multiplexed processors.

The process of identifying the target architecture has been dealt with in some detail in chapter 4. This chapter deals with the remaining steps.

5.2. Implementing Arithmetic Engines

The issues involved in arithmetic processor design apply equally to the design of both the recurrence processors and the peripheral processors. These processors may be so simple that only one level of definition is needed. More complicated processors can be structured using the hierarchy of operator, chip and subsystem, together with some conventions to aid assembly. Such conventions

are enumerated, with explanation, in later sub-sections.

The arithmetic processor in its "raw form" has some inputs, an interconnection of arithmetic processing elements and some outputs. The interconnection net defines the set of nodes. Each primitive input and each output is then connected to the appropriate node. The design and specification of a processor thus becomes simply the construction of a flow graph of primitives, together with the nodes associated with their inputs and outputs. The sequence of parallel operations to be carried out is assembled directly from the bit serial primitives available. Such a list must satisfy certain criteria of correctness in order to represent a valid network. (These criteria are checked for by both the simulator and the chip layout generation programs of the FIRST compiler; they include checks for: fan in, fan out, undriven nodes, and overdriven nodes.)

5.2.1. Time Tagging

Perhaps the largest task in designing a bit-serial arithmetic processor is that of accommodating latency through the various signal paths. Time tagging is a useful technique to help manage this process. A notional time tag is associated with each node in a net list. Time tags correspond to the arrival of new words at the hardware nodes. The tags are defined relative to some location chosen as the time tag origin and relative to some starting time.

The location for the system time tag origin is usually taken to be the control generator output. This corresponds to global tagging. Local time tag origins can be defined anywhere, for example at the inputs to a processor. The absolute starting time is

time zero. Alternatively, relative starting time may be used. This can be any value of absolute time when it is taken as zero locally. A time tag is relative or absolute depending on its starting time, and local or global according to its origin.

Define the time tag of a node to be the latency, measured in bit time, from the chosen origin. In other words, the time tag is the number of bit times required for the LSB associated marker of a signal to travel from its origin to the node in question. The time tag is a kind of time metric for the system. For convenience, the time tag can be measured in higher order units or even mixed units e.g., cycles, or words and bits, if desired. Because each bit-serial primitive is pipelined and has a finite latency equal to some integer multiple of the clock, or bit time, the processor nodes will in general have different time tags associated with them.

Time tags are used to keep account of data synchronisation during the design process and to help in generating the control network. Local time tags are usually used during processor design; changes of origin are easily accomplished during synthesis.

5.2.2. Further Conventions

To aid synthesis and speedy design of arithmetic engines it is convenient to adopt some further simplifying conventions for their construction.

Maximal Concurrency, Minimal Processor Latency

Firstly the organisation of primitives within a processor should exhibit maximal concurrency. That is to say, a fully

concurrent flow graph of the process is preferred over one which multiplexes elements within it. This organisation maximises individual processor bandwidth and simplifies both the control structure and the problems of synthesis. Further, an attempt is made to minimise processor latency by the judicious choice of parameters, the order of operations, and by avoiding, wherever possible, the use of "latency-expensive" primitives.

Synchronous Interfaces

The organisation of the system architecture, as given in chapter 4, is simplified if all data entering and leaving each arithmetic processor is synchronous. This is the second convention: to ensure that inputs to a processor and outputs from a processor are synchronous, corresponding to a principle of time aligning at inputs and outputs. This requires there to be equal latency from input to output paths. Thus, processors are arranged so that all inputs can be given local time tag zero and all outputs will have the same value of time tag.

This principle means that additional compensating, equalising or synchronising delay may have to be inserted into signal paths at various nodes. Clearly it is desirable to make this overall processor latency as small as possible, i.e., processors should incorporate only the unavoidable storage function associated with their pipelined computation and all other memory is implemented externally to the processor. Such a convention considerably simplifies synthesis at the possible expense of hardware optimisation. Generally the reduction possible when such a convention is not used is only marginal. Further, this convention can be relaxed

subsequently, to incorporate the optimal organisation.

Parameterisation Using Symbolic Constants

In general, each primitive has some parameters associated with it. The third convention is that in specifying processors as net lists of primitives these parameters should be retained as symbolic constants.

This may be necessary when some of the parameters (e.g. system word length) have not yet been determined and others may only have tentative initial values. This convention allows rapid re-implementation at any later stage if some of the parameters have to be changed. In particular it is useful to parameterise all synchronising delays in terms of the parameters on which they depend. Systematic adherence to this principle can save time later, especially if it is necessary to compare implementations with differing coefficient quantisation, or system word-lengths.

Arithmetic Aspects of Computation Architectures.

Once an appropriate computing scheme has been devised it is necessary to take account of its detailed arithmetic operation and the nature of the signal inputs. The context for this is set by the arithmetic representation, and by the application. It is necessary to establish, for each calculation in the computation scheme, the number of bits of quantisation needed and the interpretation of binary point position, the format or scaling, required. The system application will dictate a desired dynamic range and scaling for each input signal. Further, a knowledge of the nature of the signals involved should give the statistical distribution of

the signal within this range. From this information the initial range and resolution of signal values can be determined. Then, for each computation the required range, resolution and expected distribution of the results can be estimated. Methods for achieving this include theoretical considerations, statistical simulations, existing knowledge of similar systems, if such exist, and the method of first approximation and design iteration. In particular this analysis will cover internal bit growth, internally generated overflow conditions and possible scaling incompatibilities which can be caused by fixed length, fixed point arithmetic representation of signals and their different scalings.

One outcome of this analysis will be a target word-length at each point in the system determined by the arithmetic considerations. From these it is possible to see the constraints and targets set arithmetically on the system word-length. Later, as hardware processors are designed, the hardware constraints on system word-length will emerge from the design process. From both sets of constraints and targets the system word-length can then be chosen with some degree of optimisation.

It is essential to establish values for the range and resolution required for the evaluation of each iteration as well as the approximate signal distribution within this range. If this analysis is not done during this phase of design then, when it comes to processor implementation, certain assumptions will have to be made concerning them.

Two´s complement, fixed point format arithmetic introduces some unavoidable representation problems, as does any other system

of number representation. Firstly, there is the problem of bit growth resulting from multiplication operations. This is usually handled by rounding or truncation; where necessary, however, it is possible to use multiple precision product output, for example for accumulation of low significance bit values.

Secondly, there is the problem of bit growth arising from addition or subtraction. This can be handled in the following ways.

The system word-length can be chosen to be long enough to contain all bit growth. This can lead to inefficient use of dynamic range, but in bit serial systems it can also be useful where processor latency constraints and algorithm type necessitate the use of a long word-length. A variant of this solution is to catch the "overflow" in a separate higher order word, thus using multiple precision representation in the parts of the system where needed. This can reduce system word-length and speed throughput. The overhead of going to multiple precision arithmetic is small in bit serial systems, in contrast to bit parallel systems where the overhead of multiple width parallel buses is usually unacceptable. Knowledge of signal distribution within representation range is of particular value here, as it can be used to keep the probability of overflow arbitrarily small, while at the same time minimising the range needed.

An alternative approach is to detect overflow and either clamp or warn of its occurrence. This technique can introduce non-linearities into the system, and this may or may not be acceptable for some applications.

Another alternative is to ensure that overflow does not occur by scaling signal values internally, wherever necessary. A general theory for doing this in the design of digital filters is primarily due to Jackson, and a good presentation can be found in chapter 5 section 12 et seq., of [Rabiner and Gold].

The general effect of rounding and scaling is to loose precision and introduce noise into the system. For this reason an evaluation of any implementation should investigate its noise performance, that is the comparative degradation in performance over the theoretical algorithm (infinite precision) due to the way the implementation uses rounding, truncation, and scaling.

A third problem is that of fixed point format compatibility. If two signal values are to be added or subtracted, then they must have the same scale factor for the operation to be valid. On the other hand multiplication can accommodate differently scaled data and coefficient. Thus within a system some signals, e.g., product outputs, may have incompatible scaling with respect to others, to which they must be added for example. This requires careful "accounting" of scale factor, or format, at the input to each arithmetic operation, and may necessitate the introduction of scaling adjustment at various points.

Signal Scaling

The fourth convention is to label out each node with an associated signal scale factor. The FIRST target hardware and architecture are built around a fixed word-length and fixed-point format, twos-complement number representation. Each node, at each word time, has an associated binary integer value. This takes the

form of a bit-serial stream, marked by an associated LSB pulse. This integer value has an associated notional scale factor giving the interpretation of the binary point position. Conventionally, at the system data inputs, the interpretation is that numbers are represented as purely fractional, lying between plus and minus one. As a result of arithmetic processing, however, and because other inputs often require scaling, the internal representations may require to take values in excess of this limited range. The arithmetic format is expressed as the scale factor required to change the integer interpretation into the required interpretation, or in more extended form as a pair of numbers giving the powers of two represented by the most and least significant bits of the integer word (cp. Appendix II). This format then gives the assumed interpretation of the integer bit streams to be found on the node. The need for such a notation is that, given assumed input formats for a processor, it is necessary to keep track of the arithmetic transformation through the processor. For example, two signals with different scaling cannot correctly be added. Therefore if, as a result of internal operations, e.g., multiplication, two values of different format are to be added, one or both must first be scaled.

A further arithmetic consideration is that of the dynamic range (i.e., the range and resolution) and the distribution of signal values at each node of the system. This is largely a question of efficiency and use of representation. Again this is not an issue that can be resolved at the level of processor assembly but can only be dealt with at full system, or algorithm computation level. Determination of parameter values, of appropriate formats,

of system word length, dynamic range, quantisation, etc., is only possible from overall system level considerations. This issue can be resolved by theoretical studies, simulation or by design iteration.

All of the above information should be available from the analysis of the arithmetic aspects of the signal processing algorithm and its computation scheme. However, if this has not been done or is incomplete, then first approximations can be used, the system synthesised and extensively simulated and the design iterated. This constitutes an alternative to analytical or theoretical approaches.

Hardware Accounting

The fifth convention is that the specification of each processor should include a summary of both hardware and arithmetic statistics.

This means that a process of hardware and arithmetic "accounting" should complete the design of processors. This accounting includes a summary of processor function, parameterisation, hardware latency, arithmetic formats (as associated with each of the inputs and outputs), arithmetic performance, including bit growth, non-linearities, dynamic range etc. This information is used in various stages of design synthesis: during processor design, network synthesis, external interfacing, for checking and simulation output data interpretation. Systematic documentation at an early stage eases later retrieval.

5.3. Setting the System Word-Length

Systems are composed of arithmetic processors and state memory. Since the memory has no effect on word growth within the system, it follows that it is possible to evaluate signal ranges and set the system word-length as soon as all of the arithmetic processors have been designed. To do this there are two aspects of arithmetic processing to be considered: the processing local to a single processor and the total processing throughout the machine, as a result of the interconnection of processors and the consequent data flow. This sub-section discusses the issues involved in determining the system word-length. Definitive values, or at worst, first approximation values for the remaining, independent symbolic constants should be available from the arithmetic study of algorithm, computing scheme and system input signals.

5.3.1. Factors

The factors which affect system word-length are:

input quantisation;

internal bit growth through arithmetic processing;

special input formats required by primitives.

Additionally and depending on the recurrence type, a further factor which may affect the system word-length is:

minimum processor latency.

In particular this is the case for fully parallel architectures, multiplexed architectures implementing type B recurrences (see chapter 4), and serial architectures.

5.3.2. General Objectives

In bit serial implementations, the system word-length or the processor latency will determine the stage sub-cycle length and hence the processor throughput rate. It is desirable to make both as short as possible, in order to maximise hardware bandwidth. Further, unless there are overriding reasons for not doing so, system synthesis is simplified by making them equal, or at least by making the processor latency an integer multiple of the system word-length. On the other hand, it is necessary to ensure that the system word-length is long enough to allow for the anticipated bit growth. These opposing requirements on system word-length are resolved by ensuring that efficient use is made of the available dynamic range. Once the architectural decisions have been made, the method for choosing system word-length is to determine firstly the values of all the lower bounds on it. Then the least value satisfying these conditions can be chosen as the system word-length.

5.3.3. Lower Bounds on Word Length

Let outputs be indexed (i,t) according to stage and sample, respectively. Let n denote the fixed, internal system word-length. Let Q denote the fixed number of bits of input quantisation. Input quantisation forms a lower bound on the system word-length,

$$Q <= n.$$

For algorithms where the output indexed (i-1,t) is required before the computation of output indexed (i,t) can commence, the processor latency, L, is a lower bound on the system word-length,

$$GLB(L) <= L = n$$

where GLB() stands for the greatest lower bound.

Even where the restriction of making L=n is not necessary there are often reasons of convenience for chosing the system word-length to have this value, or perhaps an integer multiple of it.

5.3.4. Bit Growth

The choice of system word-length should take account of internal arithmetic growth and can be used to incorporate guard bits to deal with overflow/underflow. Any system word-length must either incorporate adequate guard-bits or some other mechanism for accommodating the effects of bit growth. Internal bit growth occurs only in the arithmetic processors (e.g., adders or subtractors), no further bit growth can occur in the memory, multiplex or interconnect hardware. When considering internal bit growth it is necessary to distinguish the following types of input condition:

single pass inputs

multiple pass inputs (fixed, finite repetition)

feedback inputs (unlimited repetition)

The classification of inputs in this way is derived from the system architecture. Single pass inputs involve only explicit bit growth within the processor. Multiple pass inputs occur when there is repeated identical processing, either within one processor, by multiplexed recirculation, or by passing processed values on to further identical processors. In this case the maximum possible bit growth is a fixed multiple of the explicit bit growth which occurs within a processor. Feedback inputs, related to time averaged or accumulated data, involve potentially unlimited bit growth.

For each input the associated bit growth must be determined. A deterministic approach or more likely, a statistical one, based on expected signal distributions, can be taken. Where the processor incorporates a scheme of scaling it may not be necessary to allow for bit growth.

5.3.5. Format Restrictions

Certain primitives have input format restrictions. These must be observed for the hardware primitive to function correctly. Usually these are in the form of conditions on the sign bit; for example, the modified Booth multiplier, discussed in Appendix II requires two sign bit repetitions for correct operation.

For convenience, the system word-length is often chosen to be an even number.

5.3.6. Summary

Having determined its various lower bounds, the system word-length may then be chosen as the least possible value to satisfy all. If a long total system word-length is necessary, it may be partitioned into multiple precision bytes of shorter length. If feasible, the byte can then be redefined as the internal system word-length and processing is in multiple precision where necessary. The cost of using this facility can be an increase in system hardware latency, which is traded off against the higher bandwidth operation.

5.4. Implementing State Memory, Multiplexing and Net Synthesis.

The number of physical processors that will be needed can now be fixed, by using the proposition of chapter 4, together with the value for the system word-length. This achieves the processor-to-system bandwidth matching necessary to give minimum hardware but adequate performance. The chosen architecture determines the sequence of intermediate result computation and gives the multiplex memory loop structure (cp., chapter 4).

Having designed the processors and determined the symbolic constant values, the system synthesis is accomplished by adding the required memory, multiplex switching and interconnect. This operation should be subject to the restrictions outlined in sections 5.5 and 5.6 below, in order to secure initialisation and testability.

As a first pass at interconnect synthesis, a useful convention is to proceed as follows. Multiplexers are connected at processor inputs wherever they are needed to break loops. Compensating delays are inserted into the other input paths, as necessary. Associate with each such processor the fixed amount of "virtual" state memory required. This is the total memory required to make up the correct loop lengths needed by the algorithm (function and architecture). Implement this partially by the inherent processor latency and partially by the addition of physical memory to make up the remaining virtual memory. Each processor output is then connected to this physical state memory and the state memory outputs are connected to the multiplexer inputs.

The interconnection of such processor, multiplexer, memory structures to each other is then achieved by tapping outputs from the processors, or from somewhere in the state memory so as to pass

data between these units in the correct synchronisation. The interconnect scheme can then be evaluated for hardware efficiency and where expedient may be rearranged, in particular by relocating the multiplexer position in relation to the processor and memory within each loop.

5.5. Initialisation

The initial internal state of all primitive hardware on power up is indeterminate. Two types of start up cover all possible system requirements. These are :

start up where pipelines fill with valid data by propagation via normal system function and

start up from given initial conditions.

The former assumes that direct flow will flush out the initial states and therefore assumes that there is no internal feedback; the latter requires a preliminary initialisation phase to set up the correct initial conditions. In hardware terms this is achieved by ensuring that all feedback loops in the network are broken by at least one multiplex primitive which allows access for external signals. The multiplex primitive is controlled by an "event" control. This event, when valid, signifies initialisation and, when not valid, signifies run mode. The event control signal is activated during initialisation, or reset by the use of the corresponding event request input to the control generator. During initialisation the multiplex primitive breaks the feedback loop by taking an externally supplied signal, instead of the internally fed-back signal. In run mode the multiplex primitive closes the internal loop, as required for system function.

5.6. Design-for-Test

It is now widely accepted that proper consideration must be given to testing as part of the design process. Without this consideration, the parts so designed may well be difficult or even impossible to test in practice. Ideally a simple design-for-test procedure is needed; one that can be applied at design time. The test and testability issues are examined in detail in [2,3]. and the conclusion of these studies is that bit-serial systems are easy to test and that a simple design-for-test process does exist. In this section a summary of the main conclusions of the afore mentioned study is given, as far as they relate to system synthesis. This is done to ensure that the systems designed are fully and easily testable.

Primitives can be classified as amenable to random test, or not. For test purposes the system network must be broken into chains of random test amenable primitives; where a chain is defined to be a net of primitives which has no loops, that is no feed-back paths. Any primitives which are not random test amenable should be isolated from the chains so that they can be tested separately. Largely, the configuration into chains will already have been arranged for the purposes of initialisation, and is achieved by the introduction of multiplex elements. Isolation of primitives which are not random test amenable can be done by the use of multiplex elements or also by other, more efficient mechanisms. This allows external access to inputs and outputs for separate testing and will prevent non-linear growth of testing.

In summary, system synthesis should ensure that, in composing

the interconnection net, all closed loops should be configured so as to have an externally accessible point of entry (this is generally done in any case for initialisation) and all non-random-test-amenable primitives should be isolated.

5.7. Implementation of the Control Network

Reviewing progress with system design and implementation thus far, it is to be observed that the following items have been derived: a system architecture, values for all the system parameters, designs of arithmetic processors and a primary network of signal nodes connecting processors, state FIFOs and multiplexing. Each primitive in this primary network has certain control requirements. So far these exist in the design as "hanging" or unconnected control inputs and outputs at each primitive. These undefined control connections, together with the time tags associated with the corresponding signal nodes, implicitly define a secondary network of control nodes. This control network has no nodes in common with the signal network (other than supply and ground) and may be viewed as an "orthogonal" design exercise. The primary (signal) and secondary (control) networks "contact" each other in the primitives. The next task in the design process is to find an efficient implementation of this control network.

The control network can be implemented in the following stages:

Time tagging;

Modulo reduction;

Definition of virtual control loops;

Reduction by origin rotation;

Physical realisation in minimum length tapped delay lines.

Time tagging has already been dealt with in section 5.2.1, the remainder of the steps are discussed next.

There is a hierarchy of control, corresponding to the incidence of the start of a word (LSB), the start of a multiplex sub-cycle, the start and finish of an initialisation or test cycle etc., at each node. Each level of control requires, firstly, a single source node from the control generator. Secondly, variously delayed versions of each level of control are required to synchronise correctly with the primitives in the primary network of signal nodes. There are two possible approaches to the construction of these delay line networks:

1. the delay elements can be synthesised as a separate control network, which supplies all control requirements but uses none of the hardware already in the primary network.
2. the delay network can be synthesised as two subnets. One, the processor-associated-control, consists of hardware, either existing or additional, within each processor. The other, the extra-processor-control, supplies the remaining control requirements.

It may appear that there is no essential difference between these two approaches, since they relate, apparently, only to specification groupings which are flattened out by the compiler. This is not the case. The use of processor associated control can result in duplication of control delay hardware, but can also utilise any "free" hardware in the place of extra delay hardware, as in the case of the multiplier primitive, which automatically delivers

a delayed version of its input control. Separate control may be minimum in terms of delay hardware but can suffer fan-out problems and may not utilise any "free" delay associated with processor elements to replace delay elements. Each system should be evaluated so as to establish which method of implementation gives the best trade-off between hardware, fan-out and design complexity, though the margin of difference is usually small.

There are two techniques for reduction of the control network hardware which can be applied.

Firstly, due to the periodic nature of the control cycle pulses, the tapped points of the delay line will be equivalent modulo the cycle-length. This means that the maximum length of control delay line necessary for any level of control is always less than the cycle length for that level. Thus LSB control can be reduced modulo the system word-length. Usually the total latency required for the multiplex cycle level of control is less than the multiplex cycle length, so that this reduction technique is not usually useful above the LSB level of control.

Secondly, again due to the periodic nature of the control cycles, each cycle associated delay line can be reduced by rotating the origin. Recall that the origin is defined at the control generator output and is associated with the start of a new cycle. Note also that the various levels of control must be in synchronisation at all locations. Given any level of control and any origin, once the control delay line has been constructed it can be notionally extended into a closed control loop if a further end-block of delay is added so as to synchronise the final output with

the initial input. (Note that the control generator does not incorporate a facility for receiving the returned signal.) The total length of the loop is then determined by the total latency of the hardware being controlled and is an integer number of words, corresponding to the delay of a signal moving through the whole path from start to finish. Further, the loop is characterised by groups of delay (usually unequal) separated by the tap out points required by the primary signal network. This shall be called a virtual control loop as it is sensible to implement only the delay line portion up to the last tap position, and not implement the end-block. This is a useful design concept for the following reason. By rotating the origin around the loop, the unimplemented end-block can also be made to rotate. In this way, a judicious choice of origin can ensure that the largest delay in the loop is the end-block and so ensure minimum hardware.

This technique is not always applicable, but it is particularly useful for dealing with multiplexed architectures, which always have signal loops associated with the multiplex-cycle level of control. Application of rotation in this context reduces both the delay hardware required for the multiplex level of control and also that required for the associated initialisation event control net.

In passing, it should be noted that neither modulo reduction nor origin rotation are techniques that can be applied to event control, since events are non-periodic and do not have a defined cycle length. Examples of the computation of time tags, the implementation of control networks and their reduction using the techniques outlined here, are to be found amongst the case studies in

[1].

5.8. Partitioning

Until now no distinction has been made to define any physical boundaries in the system. Instead global system design has been developed, independent of such considerations. This is a viable approach for bit-serial systems, where partitioning is feasible in any number of ways due to the low overhead incurred when any signal path crosses a chip boundary.

In system design, chips are specified as containing a hierarchy of operators and primitives. The use of a hierarchy of operators eases the design task by allowing nested groupings of primitives to be specified, as appropriate to the system architecture. This feature serves design convenience and efficiency, the silicon compiler flattens out any hierarchy, and assembles the chip from an ungrouped list of primitives. Partitioning consists of determining how many and what operators to place on each chip.

Partitioning is governed firstly by the maximum feasible chip size available, which is, in turn, determined by process yield and packaging economics. Secondly it is influenced by the architectural groupings of primitives. In the early stages of design, before system synthesis has been completed, (during processor design and as a preliminary to system synthesis), chip floor plans can be generated in order to obtain likely chip sizes. This will establish what limitations are imposed on the architectural groupings of operators and primitives by the chip size restrictions. This should be done at an early stage of design because partitioning involves a latency overhead. There is a latency of one bit

involved in leaving one chip and entering another, due to the inter-chip pipelining latency. Thus, the location of a primitive on a chip sets it down inside communication boundaries. Primitives cannot be moved over these boundaries without a modification of latency, if synchronisation is to be preserved. Iteration on floor plans with incomplete system specifications allows efficient partitioning prior to and during final synthesis.

System synthesis can be done with purely architectural groupings of chip contents. If this approach is taken without any idea of resultant chip size then the resultant system may contain oversize chips, or chips with unsuitable aspect ratios. This may force a redesign. The redesign will require redistribution of primitives across chip boundaries and so entail changing both the primitives and their interconnection net. Even this is not usually a major task and it can normally be accomplished quickly. In the interest of modular design it is generally convenient to partition systems so as to include whole stages of recursive computation schemes within chip boundaries. This is exemplified by the case studies in [1].

5.9. Completion

All the major design issues have now been addressed. A system design can be completed routinely in the following stages. Some of these can be initiated from the start of design and be run alongside the stages described above; for example, during processor design it is recommended that coding and compilation of flow graphs accompany their formulation.

Node Naming

Nodes define the interconnection between primitives. For FIRST source descriptions nodes must be given names. These may be descriptive or may follow some name-number scheme. The compiler generates the name-associated numerical tags, and lists these in an output file, which is used for specifying input and output nodes when using the simulator.

FIRST Source File Writing

The FIRST source code which describes the system is written next. Typically this consists of up to one or two hundred lines of code. The process of writing this code uses the information given in appendices I & II and can be undertaken piecewise from the outset.

Source File Compilation and Debugging

The source file can be compiled incrementally as it is written, to remove syntax faults using the compiler's diagnostics. Final compilation gives the files required for chip generation and simulation.

Floor Plan Monitoring

Throughout writing the source description, and during incremental compilation, chip floor plans should be inspected, as suggested in section 5.8, in order to steer partitioning.

Preparation of Simulation Input Test Files

The approach to the generation of simulation input files

requires the system designer to write some code to generate input files with the correct format. This can be done in any high level language with which the designer is familiar and for which there is a compiler. Signal generator programs can be relatively easily written and are short, even for the generation of the more sophisticated types of signals.

Simulation

Once the complete system has been coded, and the signal input files have been prepared, the system can be simulated. Firstly, this gives full checks on the node net specified. Secondly it gives full LSB level synchronisation checking. Thirdly it gives verification of functional behaviour. Finally simulation gives sets of test vectors which may be retained for later use.

Design Modification

As a result of simulation, the design may be modified in order to remove specification or synchronisation errors or system design inadequacies, which cause incorrect performance. Revisions of the source description, for any of these reasons, can be quickly completed if design follows the guidelines set down in this chapter.

Mask Generation

Full mask geometries are generated from the final, verified system chip descriptions. These are then ready for any post-design pre-fabrication processing which may be required by the mask-making house.

System Interfaces and Packaging

In order to progress from a working chip set to a self-contained, working system, chip cards, supplies, clock generators, and external interfacing, including sample and hold circuits, analog to digital converters, RAM and ROM interfaces, etc., have to be assembled. A variety of standard configurations can be designed for all of these. It is suggested that this is best done by using a minimal number of standard parts.

Documentation

The final stage of any design is completion of the documentation. One of the advantages of the design process as set out here is that it may be self-documenting to include a design "history", which records revisions and evolutions. A complete set of documentation for a final system would include :

1. An application and system description.
2. A definition of the algorithm.
3. A definition of the computation architecture.
4. A set of hardware flow graph block diagrams.
5. The FIRST source description.
6. A set of chip floor plans.
7. Simulation diagrams.
8. Simulation input and output (for test vectors).
9. Bonding diagrams.
10. Description of system interfaces.
11. System user documentation.

All of this material is generated during design. In particular, items 5-8 are "unavoidable" documentation, in the sense that

files containing this information are either required for or generated by the FIRST compiler and are in "human readable" form. If a flow graph graphics pre-processor software package is used then the flow graphs (item 4) also become a part of the unavoidable documentation. In this way most of the documentation becomes an integral part of the design activity and is not a chore to be completed afterwards. The revision and evolution "history" will consist of old versions of files and cover items 5-8. If the system designer also generates concise verbal explanations to accompany these files and cover the remaining items in the list, then the design is almost wholly self documenting.

5.10. Summary

This chapter has proposed a specific methodology for system synthesis which may be followed to generate complex custom systems in short time scales. Further, it presents a framework from which further design automation might be researched and developed. The important areas in this field are:

Data-base or file-system management and verification. (The approach used has been file system based.)

Design documentation and history.

Extension of automation to the generation of architectures from algorithm specification.

Development of a technology independent circuit specification language and layout generators for specific technologies.

More general solutions to the problem of generating the physical design, within any one technology so as to allow a greater flexibility of layout topology and a greater degree of layout optimisation and accounting.

An integrated solution to these problems may perhaps lead to a second-generation compiler embodying algorithmic specification as input.

References

1. P. B. Denyer and D. Renshaw, VLSI Signal Processing: A Bit-Serial Approach, Addison-Wesley (to be published).

2. A. F. Murray, "On the Effectiveness of Random Pattern Self Test for Bit-Serial Signal Processors," IEEE Trans. Comp., (to be published).

3. A. F. Murray, P. B. Denyer, and D. Renshaw, "Self-Testing in Bit-Serial Parts: High Coverage at Low Cost," Proc. IEEE International Test Conf., pp. 260 - 268 (Philadelphia, October 1983).

CHAPTER 6

A SURVEY OF CASE STUDIES IN SYSTEM DESIGN USING FIRST

6.1. Introduction

In this chapter system design capture is evaluated. In this way, the design environment and the design methodology described in the earlier chapters can be assessed in terms of the range of systems that have been successfully designed.

For each system the principal features of interest are complexity, performance and design time, as well as partitioning and distribution of resources (processing, memory, control etc.). The full data associated with each system comprises:

1. a functional definition of the system
2. a performance specification for the system
3. a block diagram (flow graph) of the system
4. a FIRST HDL file
5. a set of chip mask geometry definitions
6. a list of implemented system statistics
7. a set of simulations
8. a set of test pattern input and output vectors.

As detailed description of each system lies outwith the scope of this thesis, this information is not presented in full but key points are summarised. Where available, references are given to papers which cover further details. Appendix III gives a detailed presentation of items 1-4 & 6 for one system.

The system designs were carried out by a number of researchers. P. B. Denyer and the author acted as consultants during the

process of initial implementation in order to introduce each designer to the methods and tools. The extraction of system statistics and the survey presented in this chapter are the sole work of the author, as was the automation of data extraction.

6.2. System Statistics

The system statistics available are extracted as a by-product of compilation from the FIRST IDL files and include:

system wordlength

system multiplex cycle periods

primitive count

transistor count

distribution of resources (processing, memory, control etc.)

pad count

concurrency factor

number of operations per second

node count

list of primitives

list of chips

chip sizes and aspect ratios

HDL and IDC file sizes

sampling frequency

number of chips

design time

preliminary system description comments in the HDL file.

The design time can be seen as a measure of the design capture and automation efficiency. Chip count and transistor count are crude measures of system complexity and partitioning. The concurrency factor is measured by the number of arithmetic processing

primitives (including formatting primitives but not memory or control) and is of architectural significance, as are the relative proportions of processing, memory and control. Pad count can be used as the basis for measuring communication bandwidth. Often the balance between communication bandwidth and processing bandwidth are of architectural interest. The number of operations per second is measured by the product of clock frequency and concurrency factor divided by system wordlength. In this respect all arithmetic operations have an equal weighting. A break down of the actual operations and their proportionate use can be deduced from the lists of primitive types. System wordlength, given the clock frequency,provides a measure of word rate; multiplex cycle length provides information on the architectural use of processors. Node count provides a raw measure of the network complexity. Chip sizes (given in lambda units) provide an indication of practicality of partitioning and of implementation cost. Finally, the size of HDL and IDC files can be used as a measure of source data compactness.

6.3. Pilot Studies

Four systems were chosen as pilot studies. The pilot studies were completed during approximately four months at the end of 1981 and beginning of 1982, while the design environment was being constructed and before it had been completed. Their purpose was to identify problems in using the HDL, establish that useful systems could be specified using it and test and develop the methodology. The studies started from functional and performance specifications and ended with HDL descriptions and compiled chip floorplans. They were conducted independently by P. B. Denyer and the author and then in collaboration. The systems chosen were a programmable

transversal filter, an adaptive transversal filter, an adaptive lattice filter and a programmable filter based on cascading biquadratic filter sections.

As a second phase, guest systems-designers were invited to implement a system in collaboration with P. B. Denyer and the author for the purpose of evaluating the approach and facilities. This collaboration phase led to a reworking of the above mentioned systems together with the implementation of other systems. Each system has been fully implemented as HDL, compiled and verified by system simulation. Further, fine-tuning of implementation trade-offs by redesign at an architectural level has been demonstrated. The rapidity with which this can be done using FIRST opens up custom silicon design for the prototype bread-boarding of systems. In the following sections of this chapter each system is reviewed.

6.4. Complex to Magnitude Converters

This system was developed as a final year student project by D. J. Talbot. Two versions were implemented (3% and 1% accuracy algorithms). Full details can be found in [1]. The system wordlengths used were 14 and 16 bits respectively, giving single chips with transistor counts of 1.25K and 2.19K. The system was configured to operate with no multiplexing (multiplex cycle period of 1 word). Maximum system word rates and sampling frequencies were therefore equal, being 570 KHz and 500 KHz respectively. The chip sizes were 1012x1103 and 1621x1209. Proportions of arithmetic, memory, control and formatting were 82%, 7%, 11%, 0% and 83%, 7%, 10%, 0% respectively. The 3% system operated at 4.6 MOPS and the 1% system at 9.5 MOPS. Design time is estimated at 1 man-

week, and the system specification files in FIRST HDL comprised 70 and 80 lines of code. Simulation confirmed performance to better than 3% and 1% accuracy for each respective system. The 3% version was submitted for fabrication.

6.5. Finite Impulse Response Filters

This system was developed by S. G. Smith. Two versions were implemented, a real arithmetic 512 point version and a complex arithmetic 128 point version (2 channels). The system wordlengths used were both 14 bits. Chip counts were 10 (3 designs) and 18 (3 designs) respectively, with transistor counts of 240K and 162K. The systems were configured to operate with multiplexing multiplex cycle periods of 32 and 8 words). Maximum system word rates were both 517 KHz and maximum sampling frequencies were 40 KHz and 71 KHz. Proportions of arithmetic, memory, control and formatting were 11.6%, 68.6%, 19.2%, 0.6% and 56.7%, 31.6%, 10%, 1.7% respectively. The real arithmetic system operated at 93.7 MOPS and the complex arithmetic system at 246.9 MOPS. Design time is estimated at 1 man-week (including both versions), and the system specification files in FIRST HDL comprised 140 and 200 lines of code. It should be noted that each design was extensible by cascading additional chips. The system was simulated but not prepared for fabrication.

6.6. Adaptive Lattice Filter

This system was developed by M. J. Rutter, D. Renshaw and P. B. Denyer. Details can be found in [2,3]. Only one version was implemented. This was a custom IC re-implementation of an existing TTL design. The system wordlength used was 26 bits. The system

had a total chip counts of 5 (5 designs) and a total transistor count of 30K. The system was configured to operate with a multiplex cycle of period 16 words. Maximum system word rate was 307 KHz and maximum sampling frequency was 19.2 KHz. Chip sizes were 2020x2086, 2174x2129, 1768x2093, 1348x1845 and 2076x1955. Proportions of arithmetic, memory, control and formatting were 43%, 29%, 13%, 15%. Operating performance was 10.2 MOPS. Design time was 1 man-month and the system specification files in FIRST HDL comprised 500 lines of code. Algorithmically, this was the most complicated system implemented. Simulations were completed but mask geometries were not prepared for fabrication.

6.7. Echo Canceller

This system was developed by S. G. Smith, D. Renshaw and P. B. Denyer. Details can be found in Appendix III. Four versions were implemented. The system wordlength used was 18 bits. The system had a total chip count of 18 (3 designs) and a total transistor count of 252K. The system was configured to operate with a multiplex cycle of period 39 words. Maximum system word rate was 444 KHz and maximum sampling frequency was 11.4 KHz. Chip sizes were 2167x1561, 2111x1867 and 1635x1103. Proportions of arithmetic, memory, control and formatting were 13%, 84%, 1%, 2%. Operating performance was between 54 MOPS and 59 MOPS. Design time was 1 man-month and the system specification files in FIRST HDL comprised between 200 and 250 lines of code. Algorithmically, this system required the use of idling cycles. Simulations were completed but mask geometries were not prepared for fabrication.

6.8. Wave Digital Filters

This system was developed by N. Petrie, with assistance from D. Renshaw and S. G. Smith. It implements wave digital filter elements in the first version as separate parallel and serial three ports and in the second as a single, general purpose component, the universal adaptor, which can be configured to serve most functions required in building wave digital filters. Details can be found in [4,5,6]. Two versions were implemented. The system wordlengths used were 20 and 28 bits. The first system required two chip designs one each for a serial and parallel three port adaptor. The second system was a single chip system. In both a control chip was used only for simulation purposes. Transistor counts were 9K and 7.5K respectively. The system was configured to operate with a multiplex cycle of period 2 words). Maximum system word rates were 400 KHz & 285 KHz and maximum sampling frequencies were 200 KHz & 142 KHz. Chip sizes were 1880x1341 and 2629x1596. Proportions of arithmetic, memory, control and formatting were 82%, 13%, 5%, 0% and 86%, 3%, 12%, 0%. Operating performance was 13.2 MOPS and 13.7 MOPS. Design time was 1 man-week and the system specification files in FIRST HDL comprised 180 and 100 lines of code. Simulations were completed but mask geometries were not prepared for fabrication.

6.9. Systolic Discrete Fourier Transform Engine

This system was developed by G. H. Allen, with assistance from D. Renshaw and S. G. Smith. It implements two versions of a systolic real arithmetic discrete Fourier transform machine. Details can be found in [7]. Two versions were implemented, one based on a

four multiplier butterfly, and the other on a reduced three multiplier structure. System wordlengths were 24 and 28 bits. Both systems required 34 chips (3 designs), to implement a 32 point transform, unmultiplexed, or the same number of chips multiplexed to implement 32 stages in order to evaluate a 1024 point transform. Transistor counts were 151K and 140K respectively. Maximum system word rates were 333 KHz & 285 KHz and the time required to compute one transform was 0.96 milliseconds & 1.12 milliseconds for the unmultiplexed systems. Chip sizes were 537x1185, 963x1280 2125x2015 and 537x1185, 963x1340, 2528x1580. Proportions of arithmetic, memory, control and formatting were 82%, 14%, 4%, 0% and 77%, 17%, 6%, 0%. Operating performance was 88.0 MOPS and 112 MOPS. Design time was 1 man-week and the system specification files in FIRST HDL comprised 140 and 160 lines of code. Simulations were completed and mask geometries were prepared for fabrication for the four multiplier version. A plot is shown in Figure 6.1

6.10. Lossless Discrete Integrator Recursive Ladder Filters

This system was developed by L. E. Turner, with assistance from D. Renshaw and S. G. Smith. It implements 3-rd and 5-th order ladder filter sections based on implementation using the Lossless Discrete Integrator. Details can be found in [8]. The system wordlength was 22 bits. Each was a single chip filter system. Transistor counts were 4K and 7.5K respectively. The system was configured to operate with a multiplex cycle of period 2 words). Maximum system word rates were 363 KHz and maximum sampling frequencies were 181 KHz. Chip sizes were 2881x1425 and 3105x1484. Proportions of arithmetic, memory, control and formatting were

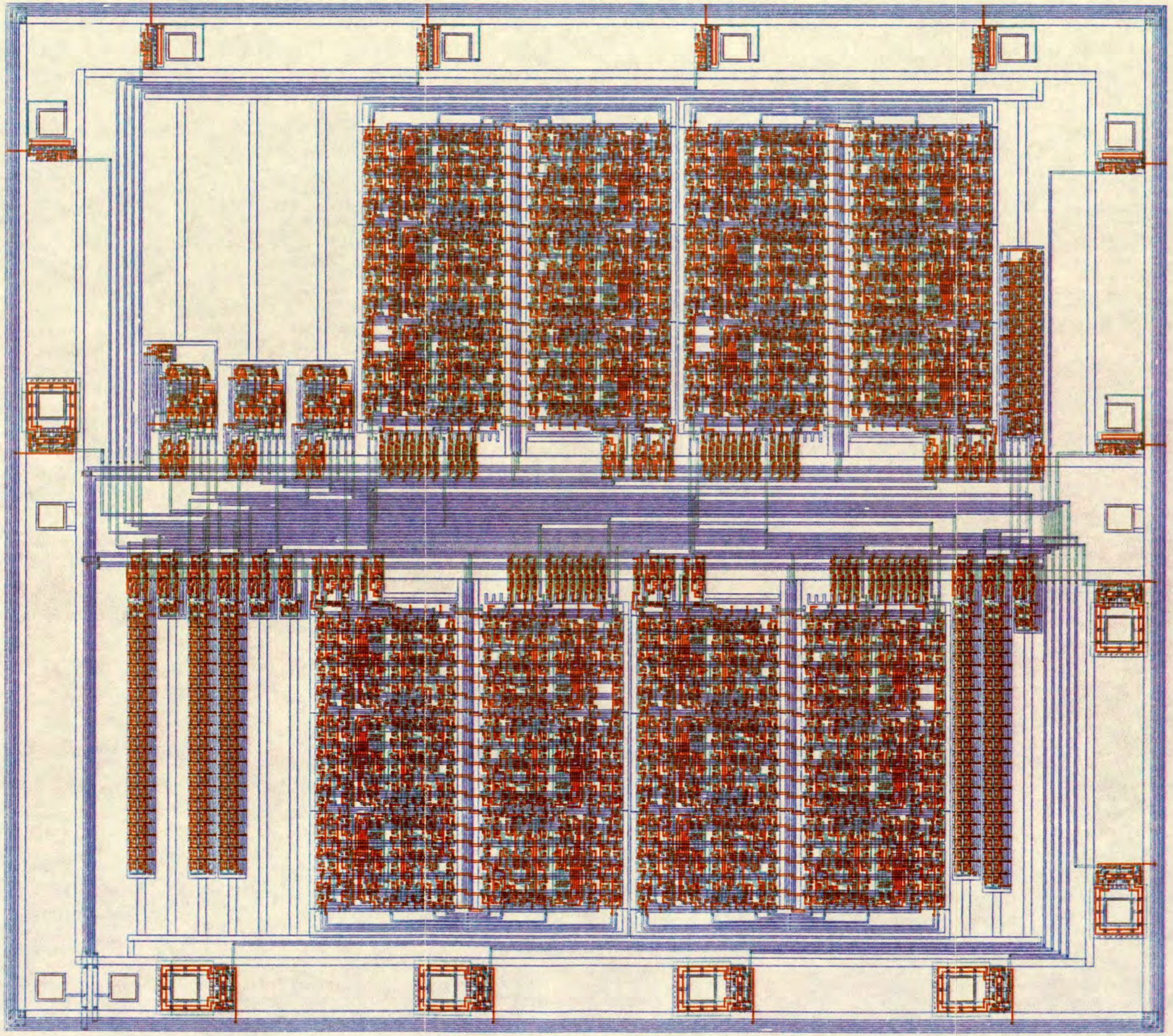


Figure 6.1

72.5%, 7%, 20.5%, 0% for both. Operating performance was 7.6 MOPS and 15.3 MOPS. Design time was 1 man-month and the system specification files in FIRST HDL each comprised 160 lines of code. Simulations were completed and mask geometries were prepared for fabrication of the 3-rd order filter section. A plot is shown in Figure 6.2. During this project a preprocessor was written, which accepts arbitrary filter specification from order 2 to order 10 and generated the required FIRST HDL. This demonstrated that special purpose very high level preprocessors are practical for special applications.

6.11. Fourier Transform Engines

These systems were developed by S. G. Smith, as a practical learning exercise for studying the FFT and DFT. Four versions were implemented. These were a 16-point, parallel, constant-geometry, decimation in time, radix 2 FFT system, using complex arithmetic; a 64-point, pipelined, Cooley-Tukey, decimation in frequency, radix 4 FFT system, using complex arithmetic; a 3-point radix 3 DFT system based on [9]; and a 6-point radix 6 DFT system, based on [10], using a complex number arithmetic based on the cube roots of unity; The system wordlengths used were 24, 20 and 20 bits. The parallel system was a practical design exercise, the others were simulation exercises, where partitioning into sensible chip sizes was ignored. Transistor counts were 160K, 87K, 15K and 35K, respectively. For the parallel engine the word rate was 5.3 MHz and the time for one transform was 0.1 microseconds. For the pipelined engine the word rate was 1.6 MHz and the time for one complete transform was 10 microseconds. For the parallel system 43 chips (6 designs) were required with chip sizes 2083x1994, 516x925, 715x1769, 1467x1329,

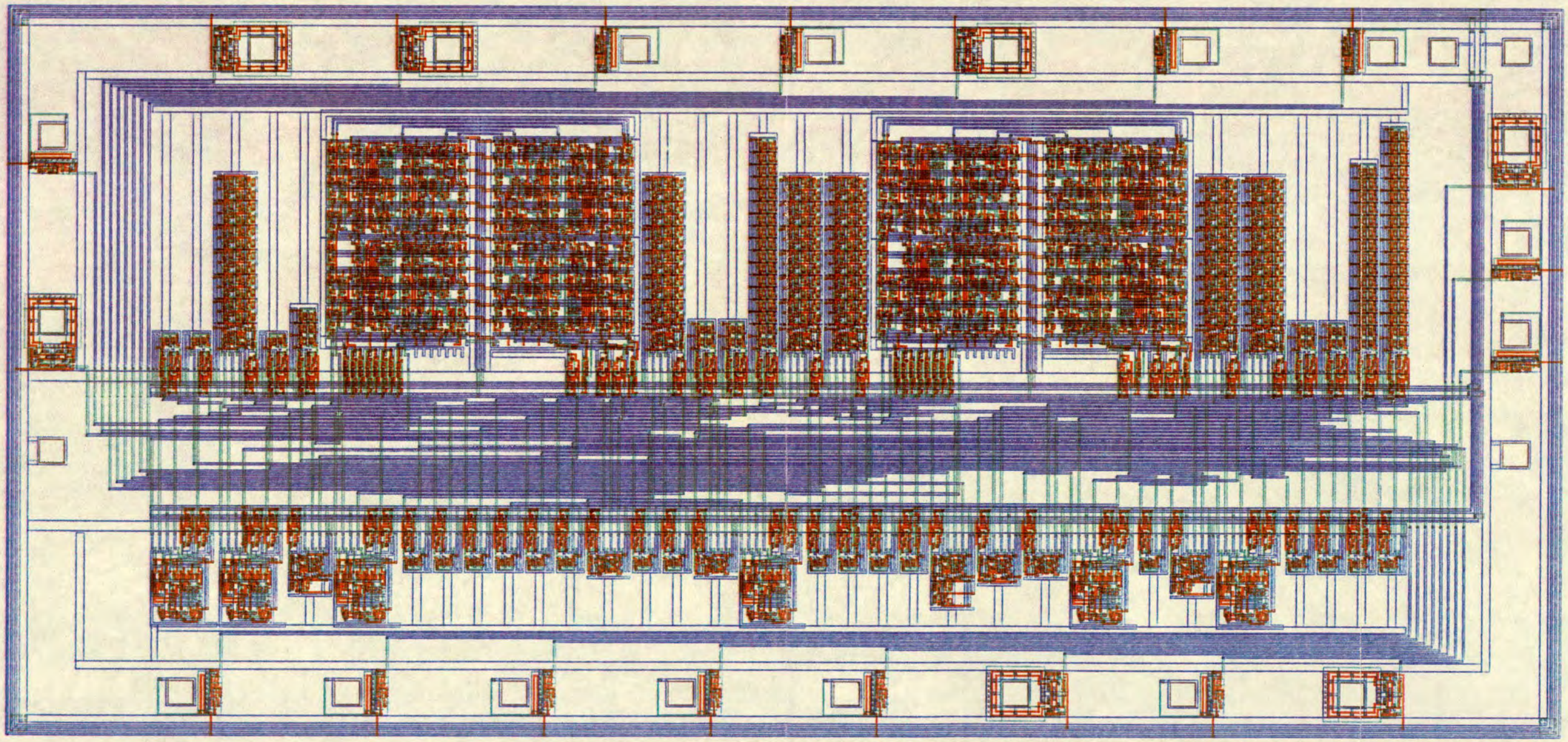


Figure 6.2

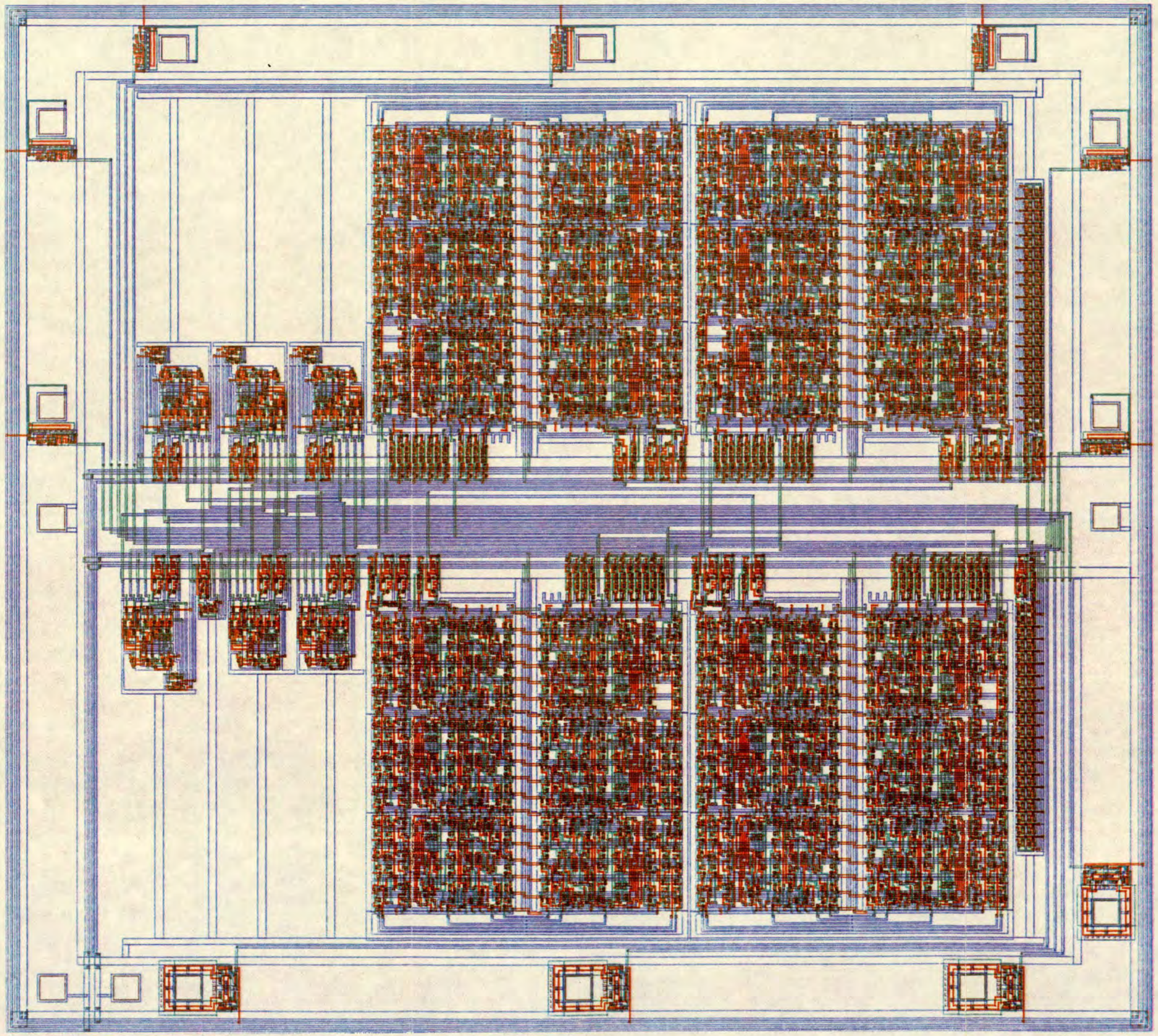


Figure 6.3

are that, if design capture and mask geometry generation are supported by rapid, cheap access to silicon fabrication using a proven cell library, then silicon bread-boarding and prototyping of complex digital signal processing systems is a practical proposition. Furthermore, design and redesign at an architectural level, with immediate feed-back on the implementation costs and implications has been demonstrated. The full advantage can be taken of freedom from lengthy delays in obtaining such feed-back on design, together with the elimination of unnecessary and excessive detail at the transistor level, which have been features of previous methods of custom design. It has been demonstrated that these tools are capable of design capture for systems of design complexity, requiring in excess of quarter of a million transistors. The success in this respect promises that the prospect of the one million transistor system on a chip can be a practical possibility from the point of view of design.

References

1. D. J. Talbot, An Integrated Complex to Magnitude Converter, University of Edinburgh, Department of Electrical Engineering (May, 1982). Project Report HSP 320

2. M. J. Rutter, , University of Edinburgh, Department of Electrical Engineering (Sept, 1983). Phd Thesis

3. M. J. Rutter, P. M. Grant, D. Renshaw, and P. B. Denyer, "Design and Realisation of Adaptive Lattice Filters," Proc. IEEE ICASSP'83, pp. 21-24 (Boston, April 1983).

4. H. M. Reekie, N. Petrie, and J. Mavor, "An Automated Design Procedure for Frequency Selective Wave Filters," SARAGA Colloquium on Electornic Filters, IEE, (March 1983).

5. H. M. Reekie, J. Mavor, N. Petrie, J. Mavor, and P. B. Denyer, "An Automated Design Procedure for Frequency Selective Wave Filters," ISCAS '83, (1983).

6. H. M. Reekie, N. Petrie, J. Mavor, P. B. Denyer, and C. H. Lau, "The Design and Implementation of Digital Wave Filters using Universal Adaptor Structures," CRISP, IEE [illegible]., Special Issue, (in preparation).

7. G. H. Allen, P. B. Denyer, and D. Renshaw, "A Bit-Serial Linear Array DFT," Proc. IEEE ICASSP'84, pp. 41A.1.1 - 41A.1.4 (San Diego, March 1984).

8. L. E. Turner, P. B. Denyer, and D. Renshaw, "A Bit Serial LDI Recursive Digital Filter," Proc. IEEE ICASSP'84, pp. 41A.3.1

- 41A.3.4 (San Diego, March 1984).

9. E. Dubois and A. N. Venetsanopoulos, "A New Algorithm for the Radix-3 FFT," IEEE Trans. ASSP Vol. ASSP-26 pp. 222 - 225 (June 1978).

10. S. Prakesh and V. V. Rao, "A New Radix-6 FFT Algorithm," IEEE Trans. ASSP Vol. ASSP-29 pp. 939 - 941 (August 1981).

CHAPTER 7

HARDWARE FABRICATION AND TEST

7.1. Introduction

This chapter describes some hardware results of design. The exercise of fabricating and testing designs was organised to provide redesign information and verification for the cell library as well as an evaluation of the design environment which has been developed and set out in the previous chapters. Originally a programme of two or three fabrication runs per year was envisaged. These were planned to take place during the the second and third years of the project. The first two or three runs were intended primarily as cell library test runs for redesign and verification purposes. Subsequent runs were seen as a vehicle for prototype system evaluation. Problems were encountered with the fabrication support substructure, however. Thus only the results of the first fabrication run are described here.

7.2. FIRST Hardware Assumptions

FIRST hardware for systems is designed on three main assumptions. Firstly it is assumed that two phase non-overlapping clocking ensures valid bit-serial data communication. Secondly it is assumed that primitive hardware implements the primitive functions that are defined in the simulator. Thirdly it is assumed that the arbitrary composition of functioning cells which have functioning interfaces gives rise to a functioning design. This third assumption will be referred to as the composition principle.

The first assumption is well established subject to the limitations of clock distribution [1]. The second assumption requires verification. This is the objective of the primitive design testing programme described in this chapter. The third assumption is apparently reasonable. In practice, however, it is true only subject to the elimination of unpredictable interactions at interfaces and to their adequate tolerancing. The assumption, however, governs composition and communication in FIRST compiled layout in a fundamental way. Thus, in cell design and testing full account must be taken to ensure that nothing done there will cause this composition principle to be violated.

7.3. Verification of Primitives

Both system assembly from primitives and primitive assembly from leaf cells is carried out on the above mentioned principle of composition. The design of primitives must therefore take account of this. Design is conducted so as to minimise indeterminacies both at interfaces and internally, across the range of processing outputs and operating conditions. The object of testing is to demonstrate that this has been achieved. The degree to which it can be achieved is largely a function of cost and time. In practice the best that can be done is to reduce the probability of failure and construct specific-case, existence proofs. It is possible that the assumption will be refuted for some cases. In such cases it is necessary to be able to determine the causes and mechanisms of failure, so that testing can yield diagnostic information for redesign.

It is necessary to define a test strategy. There are two main

alternatives. Firstly each leaf cell circuit can be fabricated and tested for functionality. Secondly primitives can be constructed, by the compiler, choosing parameter values so as to create a test cover.

The first strategy is confronted by several problems; each leaf cell has to be buffered (in order for it to drive an output) and therefore the interface cannot be observed directly. The leaf cell library contains approximately 100 elements each with, on average, around 3 inputs and 2 outputs (excluding the supplies); thus any testing will be severely limited by the consequent input and output bottle-necks. In addition, the compiler assembly and composition is not being used or tested in such strategy.

The second strategy has the advantage of testing the compiler assembly and composition output. It also tests leaf-cell assemblies interfacing in the environment in which they are designed to function. The problem associated with this strategy is how to define an adequate and cost effective test cover. A full test of all primitives with all possible parameter values is prohibitive. A reduced set immediately gives less than 100% cover. However, a cost effective reduced set can be constructed by a judicious choice of the parameter value set for each primitive. This, if properly chosen, will give only very slightly reduced probability of undetected primitive failure. The cover chosen tests all leaf cells, and all unrepeated interface combinations for each primitive in the set. Additionally this test organisation leads to an affordable primitive test programme.

7.4. Testing

The following modes of testing are normally undertaken in connection with I.C. design.

functional testing

performance testing

yield testing

parametric testing

reliability testing

In the context of the development of FIRST, however, only functional testing can be addressed adequately. This is so for the following reasons. Design for performance is done on the basis of full, guard-banded, electrical parameters. These are supplied by the processing house. SPICE [2] models are then tailored on an empirical basis to match processing. The processing offered through SERC is not supported in this way. There is no guarantee to meet the provisional values given. Further, process characterisation information does not include all the parameters. Thus "back-substitution" of actual values in place of assumed values and resimulation, though time-consuming, is not even possible. For these reasons no particular effort has been expended on trying to characterise performance.

Yield testing is an important exercise in developing a good cell library. To include this in the test programme is, however, not possible because it presupposes adequate yield characterisation of a stable process. Without this, it is not possible to characterise and separate design-yield factors from process-yield factors. In the areas of yield and performance there is no reference from which to calibrate. Additionally, much larger sample sizes

would be needed for significant results.

The problems associated with performance testing are relevant to parametric testing. Although parametric testing could have been undertaken it is argued that, against the time and effort spent, this would only be of value if the process were fully characterised.

Reliability testing requires resources which exceed those available to both the research programme and the Department. For this reason it was not undertaken.

Thus the objective of the test programme is to cover functional testing of hardware in order to verify the primitive cell library as assembled by the compiler. Included in this there must be a component capable of deriving diagnostic information about the causes of failure, where faulty cells are discovered. The purpose of this is to allow redesign to be undertaken. Yield and performance measurements cannot be interpreted for the reasons already given above.

Test Pattern Generation

Test pattern generation is simply accomplished as a result of the design-for-test methodology which has been developed. Short pseudo-random input sequences are generated, by a simple computer program. These constitute the input test vectors. They are input, with the test chip specification, to the simulator. The resulting output patterns constitute the expected test vector outputs. Thus test pattern generation is automated simply as part of the design process.

7.5. Organisation of Primitive design and test.

It was decided that testing should be organised as follows. The fundamental set of primitives were designed as quickly as possible at the start of the project and were compiled into a covering set of test chips. It was necessary to do this to allow enough time for fabrication, testing and redesign cycles within the time scale of the project.

Initially the design environment for constructing integrated circuits included only a graphics editor, geometric design rule checker, mask post-processors and SPICE. During the course of the project, a variety of switch level and timing simulators [3,4] and latterly a circuit extractor were added [5]. The availability of such tools, together with automatic translation, from the outset would have given a very high probability of success for first time cell designs. However, since a substantial part of the project had to run without such tools, it was reckoned that at least one redesign cycle would have to be incorporated, as a necessary part of the process of obtaining a set of fully functioning primitives.

Thus, at the outset, a programme to design, assemble and test an initial, reduced set of primitives was organised as phase 1 of commissioning the primitive library. Phase 2 was planned to cover redesign together with trial system design. In addition, a period of evaluation was planned for the end of each phase. The remainder of this chapter outlines the activities and results obtained during these phases of design and testing.

7.6. Phase 1

Cells for the following primitive functions were designed for phase 1.

Pads
I/O buffers
bitdelay
multiplex
add
subtract
multiply

control generator

worddelay

dshift
absolute
order

The primitives are grouped according to designer, the author being responsible for the first group and assisting in supervising students who designed the other groups. Within groupings the primitives are ordered in ascending order of complexity.

On conclusion of leaf cell design the primitives were incorporated into the compiler. This involved setting up a cell library and writing, testing and debugging composition routines. Thereafter the test cover was defined and test chips generated. The test chips were documented and submitted for fabrication. On return they were tested.

The pads and input output buffers were incorporated into earlier designs and were already known to function. The control generator was designed and tested by Neil Henderson [6]. Worddelay was designed by H. S. C. Wallace [7]. Dshift, absolute and order were designed by D. J. Talbot [8]. The remaining primitives were

designed by the author who also wrote, tested and debugged all the composition routines. Checking, correction, chip generation, submission and testing as well as redesign was also done by the author.

Pre-submission checking revealed a number of layout and functional faults in the leaf cell designs for dshift, absolute and order. These were remedied and the modified circuit logic checked and used in place of the original designs. Between design submission and the return of fabricated parts the first switch level simulator became available. An extensive set of switch level simulations of the multiplier were done in connection with investigations into testability and self test[9,10]. These revealed a fault in the multiplier composition and relative output latencies. This knowledge lead to a modification of the approach to testing the multiplier. The known faulty configuration was tested against the fabricated part to check for any further undetected errors.

7.7. Phase 1 Designs

The following primitives were tested in the first phase.

absolute
add
bitdelay
cbitdelay
multiplex
multiply
order
dshift
subtract
worddelay

Further to this, a simple trial system using instances of some of these was also included for test.

complex to magnitude 3 percent

Definition of Test Cover for Each Primitive

A test cover of a primitive is defined to be a list of instances of the primitive with the primitive parameters so chosen that all leaf cell components occur at least once. Clearly this does not test all possible instances of a primitive, which is not practically feasible, but it does ensure that all leaf cells for the primitive are tested for correct function, and that they combine to give correct primitive function. Further, because of the highly constrained way in which leaf cells are assembled and communicate, it is argued that this organisation is justified.

Table 7.1 lists, against each primitive, the parameterisations necessary to give a test cover for each.

Primitives

```
     ABSOLUTE[i]     i=6,8
ADD[i]          i=1,2,..,9
   BITDELAY[i] i=1,2,..,5
   CBITDELAY[i]      i=1,2,..,5
MULTIPLEX[i]    i=1,2,..16
MULTIPLY[i]     i=6,8,10
ORDER[i]  i=4,6,8
SCALEFIXED[i]  i=1,2,3,4
SUBTRACT[i]     i=1
WORDDELAY[i,j] where :
          (i,j)   {(15,8),(16,8),(17,8),(18,8),(19,8),
                   (16,9),(18,9),(17,10),(20,10),(29,8)}
```

Table 7.1

Partitioning

The organisation of layout was governed by the following criteria. Each operator was configured to be independently testable, including an independent power supply. Common inputs were used to

minimise pin count, maximise the number of primitives per chip and minimise the total number of chips needed. Chip sizes were arranged to fit in 5.08 mm standard EMF frames [11]. The test chip floor plan is compiler generated, in all cases.

A partitioning list is given in Table 7.2.

CHIP0	complextomagnitude[14]	
CHIP1	multiply[6]	
	add[1]	
CHIP2	multiply[8]	
	subtract[1]	
CHIP3	multiplex[i]	1 <= i <= 12
CHIP4	add[i]	2 <= i <= 6
CHIP5	add[i]	7 <= i <= 9
	bitdelay[i]	1 <= i <= 5
	absolute[i]	i = 6,8
CHIP6	order[i]	i = 4,6,8
	scalefixed[i]	i = 1,2
CHIP7	scalefixed[i]	i = 3,4
	worddelay[i,j]	(i,j) = (15,8),(16,8),(17,8),(18,8)
CHIP8	worddelay[i,j]	(i,j) = (19,8),(16,9),(18,9),(17,10)
CHIP9	worddelay[i,j]	(i,j) = (20,10),(29,8)

Table 7.2

Note

For reasons of yield four chips per mask were used. CHIP0 - CHIP3 were submitted on mask-set 1, and CHIP4 - CHIP7 on mask-set 2. CHIP8 and CHIP9 will have to be submitted together with other designs at a later phase of primitive verification.

The details of chip description source files, sizes, floor plans, bonding diagrams and pin out details can be found in [12].

7.8. Phase 1 Testing

During design the test equipment had to be requisitioned and commissioned so that it would be available when needed. At the time, the Tektronix DAS 9100 series test station [13] was assessed as being the most suitable within the available budget. All functional testing and the limited performance and yield measurements were done using this equipment. Test jigs and miscellaneous items were also designed, constructed or earmarked for use.

The phase 1 designs were completed and submitted in January 1983. Devices EU260-EU263 were received in August 1983 and devices EU264-267 in September 1983. This over-long delay (caused by an upgrade-shutdown of the EMF facility) unfortunately reduced the turnaround from a hoped-for, two to three design cycles per year to just one.

Functional testing was organised in progressive stages. Initial testing was done at 1 MHz with reduced test patterns. The initial test was, then, repeated in steps of increasing frequency until failure. If these tests were successful secondary tests using the full test patterns were carried out. If unsuccessful then reconfiguration for diagnostic debugging was attempted.

7.9. Phase 1 test results

EU260 - EU263

Five bonded samples of each chip design were received for testing. Initial testing provided the following results. The

---

DAS 9100 is a trademark of the Tektronix Company.

adder on chip EU261 failed on initial test, giving unstable outputs on all samples. The subtracter on chip EU262 also failed, giving outputs similar to those of the adder, again for all samples. All the multiplexers on chip EU363 gave consistent zero outputs for all samples, independent of test input. Of the multipliers on chips EU261 and EU262 two samples out of ten were found to be functioning, on the initial reduced test inputs; the remainder exhibited obvious stuck-at faults. Because of the failure of the add and subtract primitives the system chip EU260 could not be expected to function.

Diagnostic Tests for Design Debugging

The second phase of testing was set up to establish the causes of failure and acquire and verify redesign information. The results were as follows. Initial inspection of the adder and subtracter test outputs indicated that some kind of race hazard might be the cause. By manually circuit extracting the layout it was found that the internal carry feedback had been wrongly connected. The feedback was supplying the inverted carry prior to a phi1 clock control, resulting in both faulty logic and a race condition. These adder and subtracter faults were traced to mistaken translation of circuit schematic into layout; the circuit schematic had been correct.

It was seen from the analysis that both the adder and the subtracter could be reconfigured to test their logic and to check that there were no further errors which might have been overlooked. This was done by hard-wiring the c1 control to the supply voltage Vdd and so suppressing the internal carry. By supplying appropri-

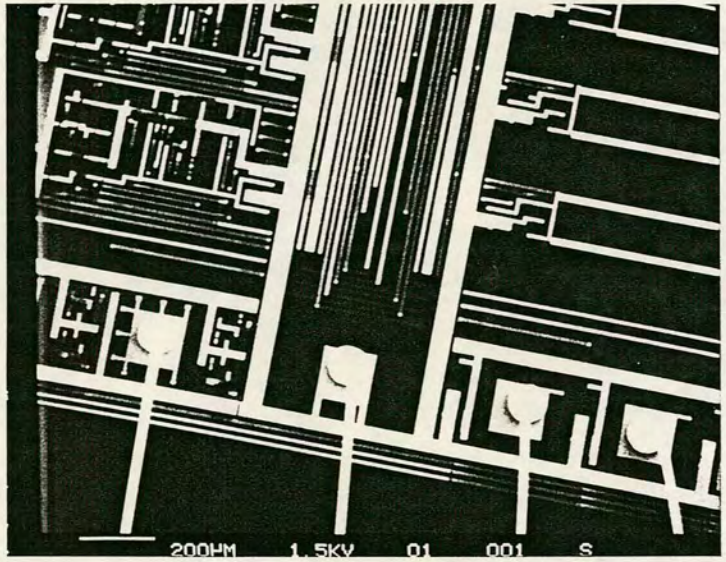
ate external versions of the carry with the inputs, both adder and subtracter were tested and were found to give the expected outputs. In this test configuration the adder circuit gave a yield of 4 samples out of 5 and the subtracter 2 samples out of 5 functioning correctly.

The multiplexer output was clearly indicative of a fatal stuck at fault. The same layout fault was found in two separate leaf cells. The leaf cells involved were the two alternative versions of the output stage and both had the ground connection of the same transistor source missing, resulting in the observed stuck at fault. Further, one of the cells had an open circuit in the ground supply. Since the stuck at fault occurred in a circuit element which was in series with the output, it was fatal and no further testing or observation was possible. No performance or yield measurements were possible either. Again, as in the case of the adder and subtracter, the faults were traced to mistaken translation of the circuit schematic into layout. The circuit schematic had been correct.

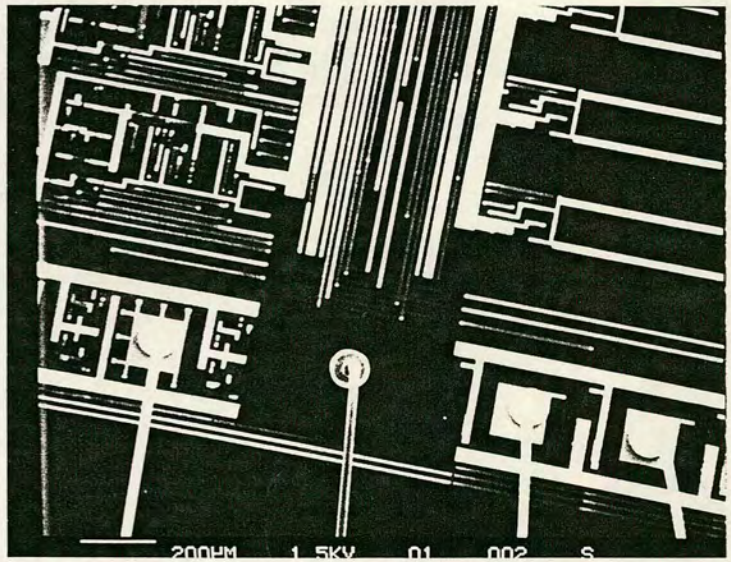
EU264 - EU267

Five bonded samples of each of the designs EU264 - EU267 were received. On testing it was found that all outputs were permanently low. Further, it was found that none of the chips dissipated any appreciable power. Initially postulated causes for this behaviour were: power to ground supply shorts, high depletion thresholds, pin mismatch on supply pad bonding or missing overglaze contacts. Visual inspection under the microscope failed to reveal any signs of short-caused burn-outs and the bonding also appeared

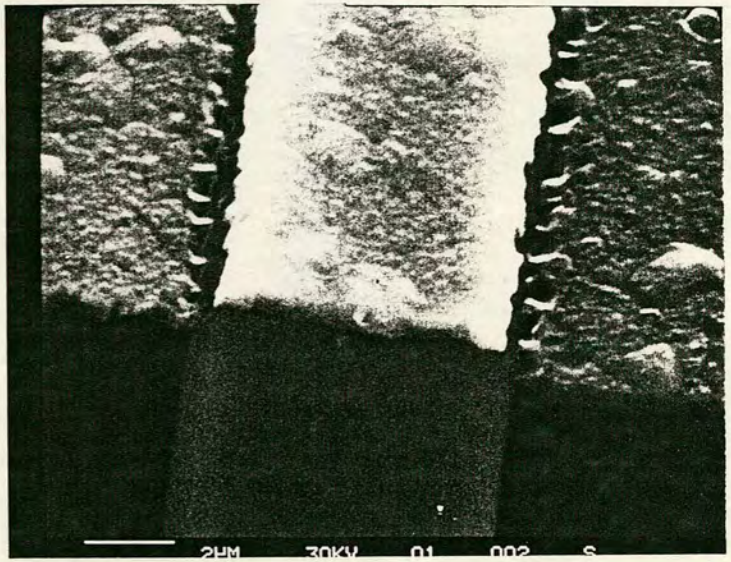
to be correct. Further, process measurements for the depletion thresholds were within the expected limits. In a previous case [14] where IC output observation was inadequate for fault diagnosis, fault mechanisms had been found by using the technique of voltage contrast scanning electron microscopy (SEM) [15,16]. Experiments were set up to examine each chip with voltage contrast under the SEM, at low magnification. It became immediately apparent that the cause of failure was discontinuity in the supply lines. This occurred in different places on the different samples indicating a processing and not a design problem. Figure 7.1 shows a SEM voltage contrast picture illustrating such sites of discontinuity; Figure 7.1(a) has 0.0 Volts applied to the centre (VDD) pad, Figure 7.1(b) has 5.0 Volts applied to the same pad, revealing the lines of discontinuity. One sample was examined at high magnification without voltage contrast at the site of a discontinuity in order to reveal the physical structure at the site. This examination was delayed till all other inspection was complete as it has to be done at a higher voltage than the functioning circuit will tolerate, and is therefore destructive. The inspection revealed bad step coverage, the break occurring at sites of metal crossing polysilicon. Figures 7.1(c) shows one such site where the break in the aluminium track is quite clear. Regrettably the whole batch was useless for functional testing because of the processing problem identified above. This meant that no redesign information could be obtained for the remaining cells. In particular it was not possible to complete the functional test of the adder or subtracter, since the delay cells used for adding latency could not be observed (chips EU264 - EU267).



(a)



(b)



(c)

Figure 7.1

7.10. Phase 1 Redesign

The redesign identified as being necessary has been outlined in the context of testing in the sections above. For reference, it is summarised here.

The add primitive had one faulty circuit element in two leaf cells. The redesign of these cells was circuit subsequently extracted and simulated.

The subtract primitive redesign was identical to that for the adder.

The multiplex primitive had one faulty circuit element in two of its leaf cells and also one geometrical design rule violation and an open circuit in the Vss supply.

The multiplier had one composition routine bug and one geometrical design fault.

Because of faulty processing, none of the other primitives could be tested, so that redesign information was not available.

7.11. Implications

The implications of the above described test results are now outlined. These fall into three categories: implications for the cell design environment, implications for the chip testing environment and implications for the FIRST compiler design method.

7.12. Implications for Cell Design

The experience gained in this and in previous projects reinforced the view that successful design depends both on having the

right design methodology and on having the right tools environment. If components of either are missing then success in design is unlikely, unless several redesign iterations are undertaken.

For the first part of the project neither a switch level simulator nor a circuit extractor were available. Nor was there automatic translation between the various levels of data representation. It was foreseen that this would cause problems and steps were taken to try to obtain both as soon as possible. Unfortunately, with the limited resources available, neither were in service for phase 1 design and submission. Thus it was accepted that phase 1 design was likely to require at least one redesign iteration. This turned out to be true. For phase 2, mixed mode simulation was available. After submission of phase 2, a circuit extractor became available, but automatic translation between representations was still incomplete.

On the separate issue of mask geometry generation and mask geometry post-processing few problems were encountered in preparing phase 1 submissions. Post-processing was done partially using the Department of Computer Science VLSI software [17] and partly using GAELIC [18] through the SERC facility. The only problem of potential concern was the partial failure of the merge program of the GAELIC post processing software.

7.13. Implications for Test

The estimates for yield gave values which, on extrapolation, looked very unfavourable. The chances of obtaining one functioning system, assuming good design and these values, on the basis of even ten bonded samples was predicted to be "infinitesimally" small for

all practical purposes. This view was consistent with the results obtained by two other researchers [19,20]. Thus the test strategy was reviewed and revised.

It was decided that, instead of accepting bonded samples on the basis of visual inspection, all wafers would be subjected to comprehensive functional probe testing. A Teledyne TAC probe test station was available in the department so it was decided to configure the phase 2 test programme around it and the Tektronix 9100 DAS, used during phase 1.

7.14. Implications for FIRST

None of the hardware test results had any implications to refute the assumptions or the overall concept of the FIRST design methodology and tools environment. The compiler was apparently problem free. There were, as anticipated, however "bugs" in the primitive cell library designs.

Postscript

Phase 2 designs have been completed and submitted for maskmaking. It is estimated that the results of phase 2 will not be available before August or September 1984. These will be reported elsewhere when available. As a separate development, an automated verification procedure has been devised using the compiler, general UNIX facilities, a circuit extractor and timing simulator. Finally, methods for the speedy design of new primitives must be explored.

References

1. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley (1980). pp. 65, 218 - 242

2. A. Vladimirescu, A. R. Newton, and D. O. Pederson, SPICE Version 2G User's Guide, University of California, Berkeley (1980).

3. K. P. Coplan, "User Guide to the Switch Level Simulator," Internal Report to SERC: "VLSI Circuits for Digital Signal Processing: First Progress Report" Appendix 4, University of Edinburgh Department of Electrical Engineering (1983).

4. L. D. Smith, "A Tutorial Guide to MIXSIM-1," Internal Report, University of Edinburgh Department of Computer Science (October 1982).

5. K. P. Coplan, "A User Guide to CEX," Internal Report, University of Edinburgh Department of Electrical Engineering (March 1984).

6. N. A. Henderson, "A Control and Timing Generator for FIRST," M.Sc. Project Report, University of Edinburgh Department of Electrical Engineering (September 1981).

7. H. S. C. Wallace, "Design and Implementation of a Word-delay Integrated Circuit," B.Sc. Project Report, University of Edinburgh Department of Electrical Engineering (May 1982).

8. D. J. Talbot, "An Integrated Complex to Magnitude Converter," B.Sc. Project Report, University of Edinburgh Department of

Electrical Engineering (May 1982). HSP 320

9. A. F. Murray, "On the Effectiveness of Random Pattern Self Test for Bit-Serial Signal Processors," IEEE Trans. Comp., (to be published).

10. A. F. Murray, P. B. Denyer, and D. Renshaw, "Self-Testing in Bit-Serial VLSI Parts: High Coverage at Low Cost," IEEE International Test Conference - Cherry Hill, pp. 260 - 268 (October 1983).

11. A. M. Gundlach and R. Holwill, Edinburgh Microfabrication Facility Design Rules: N-Channel Silicon Gate Process 2 (Revision B). March 1981.

12. D. Renshaw, "FIRST Primitive Verification Phase 1.," Internal Report to SERC: "VLSI Circuits for Digital Signal Processing: First Progress Report" Appendix 3., University of Edinburgh Department of Electrical Engineering (1983).

13. DAS 9100 Series Operator's Manual, Tektronix Inc. (November 1982).

14. D. Renshaw, "A Digital Signal Averaging L.S.I.C.," M.Sc. Project Report, University of Edinburgh Department of Electrical Engineering (September 1980).

15. E. Menzel and E. Kubalek, "Fundamentals of Electron Beam Testing of Integerated Circuits," Scanning Vol. 5 pp. 103 - 122 (1983).

16. A. R. Dinnis, "Colour Display of Voltage Contrast in the SEM," Scanning Vol. 3 pp. 172 - 176 (1980).

17. J. G. Hughes, "VLSI Design Tools," Internal Report, University of Edinburgh Department of Computer Science (1981).

18. GAELIC User Manual, Compeda (1980).

19. W. S. Blackley, M. A. Jack, and J. R. Jordan, "Built-In Test and Self-Repair Mechanisms in a Digital Correlator Integrated Circuit," NATO AGARD 47-th Symposium of the Avionics Panel on Design for Tactical Avionics Maintainability, pp. 289 - 294 (1984).

20. I. R. MacTaggart, Private communication 1983.

CHAPTER 8

CONCLUSION

The problems of implementing digital signal processing systems using custom integrated circuit technology have been presented. For VLSI, the traditional methods fail with respect to complexity management, cost, design time, debugging, and redesign cycle length. Further, system design in such an environment tends to be constrained in prototyping by TTL and standard part breadboarding. This results in mapping such prototype systems into silicon. The consequences are gross architectural inefficiency in silicon design. Thus a need for system design prototyping directly into silicon has been identified. A mechanism for achieving this must allow architectural and partitioning flexibility at the system level. At the same time it must lead to efficient silicon design. Further, the complex details of the silicon circuit design must be `hidden' from the system designer. The development of an environment which supports these demands has been the objective of this research.

A review of recent developments in design methodology and CAD has been presented. From this, important techniques have been identified for application in solving the problems identified above. A suitable class of signal processing architectures has been identified and related to the successful results of other researchers using this architecture to implement systems.

A prototype design environment fulfilling the requirements identified above has been developed. This design environment

features:

a constrained but appropriate architecture
a method for mapping algorithms to architectures
a high level language for system design capture
a compiler
bound function, behaviour & physical design
automation using modern software architectures
a minimal cell library
parameterised, procedural generation of masks
simple but efficient floor-planning and wiring
in built design-for-test and ATPG
fast simulation with bit-level accuracy

This design environment has been evaluated with respect to system-design capture, evaluation and simulation. The significant advantages of the approach are thus demonstrated. It is argued that the simplification and systematisation of design using these methods contribute towards improving dissemination and communication of design skills. Also hardware, for cell library verification, has been fabricated and tested.

Finally the problems encountered with this approach have been identified. One class is associated with the design and verification of a primitive cell library, especially in the context of re-implementing in other technologies. The other class of problems concern access to cheap, fast-turn-around silicon.

The limitations of this design environment are the constrained architecture it supports, and the technology specific nature of the

cell library together with the cost of its design and verification. It should be noted that, in a project of the type chosen, there is a balance between the relative costs and complexities of the hardware (cell library) and software components.

Scope for future work lies in several areas. Firstly, the area of cell library design and verification requires more development. Secondly, a rapid technique for mapping existing cells into other technologies is needed. Thirdly, an extension of the internal architecture to allow greater flexibility is possible. Such an extension would address several areas including partial internal parallelism, data driven control structures, greater programmability and scope for reconfiguration. Fourthly, a more general control structure should be implemented; this would provide both for architectural extensions and more flexible multiplexing.

APPENDIX I : FIRST HARDWARE DESCRIPTION LANGUAGE

ABSTRACT

This appendix describes the FIRST hardware description language (HDL). The HDL specification was jointly worked out by P. B Denyer, D. Renshaw and N. Bergmann, in consultation with I. Buchanan and J. P. Gray. The contents of this appendix were written by D. Renshaw; it appears also as a part of an appendix to "VLSI Circuits for Digital Signal Processing: First Progress Report", Integrated Systems Group, Department of Electrical Engineering, University of Edinburgh.

1. The FIRST language and its compiler.

1.1. Informal introduction.

The FIRST language allows a system designer to specify system within its scope concisely and completely, as a hierarchy of interconnected functional elements. The hierarchy of types consists of (top down) :

system
sub-system
chip(s)
operators
primitives

Primitives are the predefined functional elements of FIRST. (cp manual section 6. part III) Functional elements of all other types are user defined.

Definition of functional elements (apart from primitives) can follow top down decomposition into assemblies of simpler elements, or bottom up synthesis, or a mixture of both. However, in the formal specification all functional elements (apart from primitives) must be defined (or declared) before being used (or called) in the language description (or program). The definition of a functional element consists of a type definition statement, followed by the definition body, followed by an end statement. The definition body consists of instantiations of previously defined elements (possibly including primitives). Connections between elements are specified implicitly by using the same node name for all inputs and the one output on each node.

Instantiation of an element is achieved by calling the element. This call statement must include values for each of the

element attributes (viz.: the input & output connection names, parameter values, etc.).

In general, each type of functional element may take some or all of the following attributes :

identifier
{parameterisation}
{control inputs}
{control outputs}
{signal inputs}
{signal outputs}

where the notation { } means optional occurrence.

The identifier is the predefined primitive, or a user chosen name which is used for defining and calling the element. Parameterisation can be e.g.: the number of words or bits of storage in a FIFO memory, the number of bits quantisation for a multiplier coefficient, or the maximum count value for a control cycle, etc.. Inputs and outputs are identified by a user chosen label which is declared as the name of the node to which they connect.

Note that chips are not allowed to be parameterised.

The structure described above is embedded in the language syntax as follows :

For a declaration :

Type Identifier [Parameter list] (Control list) Signal list

For an instantiation :

Identifier [Parameter list] (Control list) Signal list

A FIRST program (or source file) can be thought of as consisting of a sequence of statements which are either declarations or

instantiations of functional elements of one of the five types.

The form of the language is closely related to the structural design languages developed by Gray and Buchanan (see Reference [79] in the reference list for chapter 2).

The function of the FIRST language compiler is to reduce the hierarchy of system, subsystems, chips and operators into a list of invocations of primitives grouped into chips, and to associate an unique identifier code with each primitive, node name and control name. The resulting description is the FIRST intermediate form or I.F., for short.

The FIRST language compiler is a single pass, four stage recursive descent compiler which takes a FIRST language program and outputs its equivalent FIRST I.F. code representation. The four stages are :

line reconstruction
lexical analysis
syntax analysis
I.F. code generation

During each stage various checks are carried out and diagnostics are issued on the occurrence of detectable errors.

FIRST I.F. code consists of a coded, expanded list of the functional element calls together with the associated interconnection net lists, as defined by the FIRST source program file. This form is defined so as to give a clean interface into physical design compilation and behavioural description compilation.

There now follows a more detailed and formal description of the FIRST source language together with an enumeration of its

features, the formal definitions.

Details of the FIRST language

Keywords
Identifiers
Types
Constants
Arithmetic
Separator characters
Expressions
Comments
Declarations : Functional elements
Constants
Signals & Controls
Scope
Calls : Functional elements
Repeated calls
Assignment replacement
Control generator instantiation
Special form of Pad instantiation
Pad order statement
Statements
Program structure
Checks, diagnostics & warnings
Formal definition of language

1.2. THE FIRST LANGUAGE :

A FIRST program consist of a sequence of keywords, identifiers and constants together with arithmetic operators and separator characters.

1.2.1. KEYWORDS

A keyword is a sequence of upper or lower case letters (which are not distinguished) forming one of the reserved words. A list of reserved words and special characters is given in table 1. Keywords must not be used for user defined names.

TABLE 1. KEYWORDS & SPECIAL CHARACTERS

SIGNAL	VDD	=
CONTROL	GND	[
PADIN	NC	]
PADOUT		(
PADORDER	EVENT	)
CONSTANT	CYCLE	,
END	PIN	;
ENDOFPROGRAM	POUT	->
CONTROLGENERATOR	PCIN	+
ENDCONTROLGENERATOR	PCOUT	-
OPERATOR		*
CHIP		/
SUBSYSTEM		
SYSTEM		

together with all Primitive names
(see section 6, part iii of this manual).

wordlength
(used by the simulator, cp. constant declaration).

1.2.2. IDENTIFIERS

An identifier consists of a string of upper or lower case letters (no distinction is made) and digits, of any length less than 256 characters, The first character must be a letter. Special

characters or spaces are not permitted. In the formal definitions <NAME> is used to denote an identifier.

Identifiers refer to primitives, operators, chips, subsystems, system, control nodes, signal nodes, constants, parameters, etc.. A list of predefined identifiers is given in table 2. As with key-words, the predefined identifiers may not be used for any other designation.

TABLE 2. PRE-DEFINED IDENTIFIERS :

VDD
GND
NC

all Primitive names (see section, 6 part III of report).

1.2.3. TYPES

Identifiers are typed and must be one of the following types :

Primitive
Operator
Chip
Subsystem
System
Constant
Signal
Control

All primitives and the three signals/controls VDD, GND,NC are pre-defined, everything else is user defined.

Note that only one identifier of type "system" is permitted to occur in any one program. Note also that VDD , GND , NC are the only nodes that can be both signal and control. All other nodes can be either signal or (exclusive or) control.

1.2.4. CONSTANTS

Constants appear as parameters which characterise functional elements (excepting chips and the system), and as operands for arithmetic expressions. Constants are integer valued.

1.2.5. ARITHMETIC

FIRST uses only integer arithmetic and supports only the following integer arithmetic operations :

( )	brackets
-	unary minus
* , /	integer multiply, divide
+ , -	integer add, subtract

Operators are listed in order of precedence, highest first, and operators of equal precedence are on the same line. Brackets are used to alter the order of evaluation. Brackets should be used to ensure correct evaluation (as evaluation of operations of equal precedence is right to left in this version, but may change in later versions).

1.2.6. SEPARATORS

Spaces, commas and newlines act as separators / terminators variously, as required.

1.2.7. EXPRESSIONS

An arithmetic expression consists of a sequence of operands separated by operators, possibly preceded by an unary operator. Operands may be integer constants, previously declared constant identifiers or expressions (sub-expression). Note that a single identifier or constant is counted as being an expression. Brackets

may be used to alter the inherent order of evaluation of an expression, by changing precedence, and should be used to ensure correct evaluations. All operations are integer arithmetic operations. Thus care must be taken in writing expressions so that the intended evaluation is actually carried out. E.g.:

Let B be any even integer then:

```
3/2*B  =  3/(2*B)
is not the same as  :
3*B/2  =  3*(B/2)
is not the same as  :
B*3/2  =  B*(3/2)
```

1.2.8. COMMENTS

Lines which start with the exclamation mark character ! are comment lines. The compiler ignores all characters of text up to and including the next statement separator (viz.: newline).

1.2.9. DECLARATIONS

All identifiers (except those which are pre defined, or are dummy identifiers) must be declared before being called. Identifiers may be declared within an operator , chip , subsystem or system declaration body, or they may be declared outside such. In the former case they are LOCAL identifiers, in the latter case they are GLOBAL identifiers.

1.2.9.1. FUNCTIONAL ELEMENT DECLARATION

A functional element (operator, chip, subsystem, system) is declared by a statement of the form :

<type><name>{<param list>}{<ctrl list>}{<sig list>}

where the abbreviations are :

param - dummy parameter
ctrl - dummy control
sig - dummy signal

and { } denote optional occurrence.

followed by a definition body, followed by an end statement.

The dummy control and signal lists are lists of dummy identifiers for signal and control nodes, which are replaced by defined control and signal identifiers, whenever the element is called. Dummy signal and control identifiers are treated as if they were local signal and control identifiers by the definition body. Similarly the dummy parameter list is a list of dummy parameter identifiers (not expressions) which act as local constants within the body of the functional element definition but are replaced on any functional element call.

The body of the definition consists of declarations of local type (which may be declarations of constants, signals, controls, or pads but may not be declarations of functional elements) and calls of predefined functional elements.

Whenever the functional element being declared is later called, each of the calls within the declaration is itself called in turn, and the dummy parameters and nodes are replaced by the corresponding actual parameters and nodes included in the call statement.

Functional element definitions or declarations MAY NOT BE NESTED OR RECURSIVE, but can call any previously defined functional

elements of any allowed type.(see note 1.).

The end statement is of the form

END

A declaration of type chip or of type system is also implicitly an instantiation of that chip or system. A functional element declaration is always global (see scope).

Operators may call only primitives and user-defined operators, chips may call only primitives and user-defined operators, subsystems may call only chips or subsystems, and the system may call only subsystems and chips. Any other structure of calls is illegal.

TABLE OF PERMITTED CALLS

Functional element	permitted calls
Primitives	-
Operators	primitives, operators
Chips	primitives, operators
Subsystems	chips, subsystems
System	chips, subsystems

Chips must include pad calls and a pad order statement in their declaration body.

1.2.9.2. CONSTANT IDENTIFIER DECLARATION

Functional elements which admit parameterisation by integer valued parameters (e.g.: number of bits delay etc.) can be called with different values in each call. In order to allow more flexible use of these parameters, identifiers may be used to name integer constants for subsequent use as parameters. Upon declaration, such

constant identifiers are given an associated integer constant value and thereafter the use of the identifier is equivalent to the use of this integer constant. These identifiers are NOT integer variables and are assigned ONCE only in the program, at the point of declaration. They cannot be reassigned within the scope of their 'lifetime'. The use of this mechanism greatly simplifies consistent parameter changes (e.g.: system wordlength, coefficient lengths etc.) in a system specification, during the functional- behavioural evaluation part of system development.

The declaration of a constant takes the form :

CONSTANT <assignment list>

where the assignment list is a list of assignments, each assignment separated by commas and of the form :

<name> = <expression>

Integer constants may be either a string of numeric characters : '0','1',...,'9' which represent integers in base ten, up to a maximum numerical value of 2147523647. Constant declarations may be local or global.

There is one constant identifier which has a special significance for the simulator. It is the name

wordlength

which is taken as the fixed internal system wordlength. It is suggested that, at the top of the source program, there is a constant declaration of wordlength for the use of the simulator. If this declaration is missing then the simulator will request it. If, however the declaration of wordlength has any other meaning then simu-

lation will be faulty.

1.2.9.3. SIGNAL & CONTROL NODE DECLARATIONS

Signal node identifiers are declared by a statement of the form :

SIGNAL <name list>

where <name list> is one or more identifiers, separated by commas. The identifiers listed are declared to be the names of signal nodes. Signal declarations may only be local (see scope).

Control node identifiers are declared by a statement of the form :

CONTROL <name list>

Control declarations may only be local (see scope).

1.2.10. SCOPE

The SCOPE of an identifier is the region of the program in which the identifier is defined and can be used. The scope of a LOCAL identifier is from immediately after its declaration to the end of the functional element in which it is declared. The scope of a GLOBAL identifier is from immediately after the point of declaration until the end of the program.

Signals and controls are local. Constants may be global or local. Primitives, operators, chips, subsystems, and the system are global. If a local identifier is declared with the same name as a previously declared global identifier (viz.: redeclared) then the global identifier cannot be accessed within the scope of the local identifier (even if the two identifiers are of different types). A

name always refers to its most local incarnation. All identifiers of the same level (local or global) must be unique.

The name of a functional element is considered to be declared after its associated end statement. Names of primitives are considered to be declared from the start of the program as are the other predefined names. Signal, control and dummy parameter arguments are considered to be local identifiers of the appropriate types, with scope from the point of declaration until the end of the functional element with which they are associated.

1.2.11. INSTANTIATION

1.2.11.1. FUNCTIONAL ELEMENTS

Functional elements are called or invoked by a statement of the form :

<name> {<param list>} {<ctrl list>} {<sig list>}

where : { } denote optional occurrence and : <name> is a primitive, a (previously declared) operator, chip, or subsystem identifier.

{<param list>} is an optional list of arithmetic expressions, separated by commas and enclosed in square brackets. The expressions, when evaluated, yield integer values which control the parameterisable attributes of the functional element. Chips and the system cannot be parameterised.

{<ctrl list>} is an optional set of control node identifiers of the form :

'('<input list> '->' <output list>')'

where <input list> & <output list> are lists of signal identifiers separated by commas. Note

{<sig list>} is an optional set of signal node identifiers of the form :

<input list> '->' <output list>

where <input list> & <output list> are lists of signal identifiers separated by commas. Again, '->' may be omitted if <output list> is empty.

The number of parameters, controls and signals associated with any functional element are fixed (except in the special case of the control generator - see section on control generator). Calls of a given element must have expressions and names which match, in number and type, the declaration. Although the lists are described as optional, using the notation { } , this refers only to the fact that e.g.: if an element has no parameters then the parameter list will be absent in its syntax. For any given element the number of parameters, signals, and controls is fixed.

Chips and the system are called implicitly by declaration. This convention is used to output I.F. code for the physical design subsystem and for simulation using the behavioural description compiler.

1.2.11.2. REPEATED CALLS

Any element, most notably chips, can be called with a fixed repetition number. The form of the repeat call is to add to the end of the call itself a statement of the following form :

"TIMES" <const> "WITH" <ctrl list> <sig list>

where the controls and signals in <ctrl list> and <sig list> are cascaded with the connection of outputs to inputs as defined by these lists, (cascaded connection). Any inputs (or outputs) which do not occur in these lists ,but do occur in the functional element call, are all connected to the same node (global connection). The assignment replacement statement must be used in order to obtain distinct inputs or outputs (see next section) from globally connected inputs or outputs. Note that outputs should never be left globally connected and must always be replaced subsequently to get distinct outputs. Only inputs can be permitted global connection.

The effect of this call is the same as having <const> separate calls of the functional element, thus giving a compact notation for repetitive identical structures. This is particularly useful for the specification of filters which may have, e.g., 32 or more identical chips cascaded, cp. appendix 1.

1.2.12. ASSIGNMENT REPLACEMENT

The assignment replacement statement is a special statement of the form :

<name> = <list>

where <name> is a signal, or control identifier, and <list> is a list of identifiers of the correct type separated by commas.

The effect of this statement is to replace each previous occurrence of <name> in the present functional element definition (within the scope of <name>) by a name in <list> . The names in list are taken in order and replace each occurrence of <name> , in

order, starting with the first and continuing the replacement down to the assignment replacement statement itself,or until the members in <list> are exhausted. The types must be the same. If the number of entries in <list> is less than the number of previous occurrences, then the remaining occurrences are left and are not replaced. If the number of entries in <list> is greater than the number of occurrences then a fault diagnostic is issued.

Usually the identifier <name> is a globally connected node in the repeated element structure. There is no access to referencing cascaded connections except into the first element and out of the last element of the repetition. Examples in appendix 1 show how various connections of repeated structure can be specified using this syntactical construct.

The purpose of the replacement statement is to allow non-global,non-cascaded inputs or outputs to be specified in a repeat statement. The replacement being of cascaded connections subsequent to a repeat statement.

The identifiers used for non-global, non-cascaded connection nodes of repeated elements should be unique, in order to avoid any unexpected side effects.

Note : It is recommended that the assignment replacement construct be used together with a foregoing repeat statement as the only statements within an operator or subsystem declaration, viz.: assignment replacement should always be embedded inside an operator or subsystem. The <name> which is replaced by the names in <list> can then be an identifier which is local to the operator or subsystem. This requirement is not

mandatory but it is strongly advised that it be observed in system specification in the interest of correct synthesis. Care should be exercised if assignment replacement is used in any other way. This restriction does not apply if the repeat statement does not have an associated assignment replacement statement.

1.2.13. CONTROL GENERATOR INSTANTIATION

The control generator construct is both a declaration and an instantiation. This is because the parameterisation of the control generator has a special, variable form. The control generator construct defines the system control. For simulation it is mandatory to have a control generator in the system. The control generator may be included, together with the system on a single chip, if this is feasible, or may be on a chip on its own or with other components for a multiple chip system.

Cycle operators indicate levels of control.

An event operator may follow any cycle operator.

Each control block has one output, except the first which also has ´inhibit´ output.

An event operator has one input and one output.

In the input/output lists at the head of the controlgenerator declaration, all inputs and outputs are in the order of the blocks below it.

The compiler gives a single block (called 'CG') in the intermediate form file, hence CG is a reserved primitive name.

A control generator must be used to generate cLSB. The first cycle statement of the control generator will have cycle length equal to system word length. This will be the value which should be declared as wordlength for the simulator.

The form for the control generator statement is as follows.

```
Controlgenerator (in1,in2,..,inx -> inhibit,out1,out2,..,outy)
  Cycle[period]
  Event
  Cycle[period]
  .
  .
  etc.
  .
Endcontrolgenerator
```

Where in1,in2,..,inx are the event request inputs.

Where out1,out2,..,outy are the cycle and event outputs, in order.

Where x is the number of event requests

Where y is the total of the number of cycles plus the number of event requests.

Where period is replaced by an integer or constant value.

1.2.14. SPECIAL FORM OF PAD INSTANTIATION

The normal syntax for instantiating a pad is a single primitive call of one of the following :

Padin(externalctrl -> internalctrl)

Padout(internalctrl -> externalctrl)

Padin externalsigl -> internalsigl

Padout internalsigl -> externalsigl

This can necessitate many pad statements in a chip definition. A shortened form, with multiple controls and/or signals is permitted, so that only one padin statement and one padout statement need be used. If a statement of this shortened form is used, then it is expanded out by the compiler to the equivalent multiple calls of padin and padout.

The syntax is as follows.

Padin(<ctrl list> -> <ctrl list>) <sigl list> -> <sigl list>
Padout(<ctrl list> -> <ctrl list>) <sigl list> -> <sigl list>

Where the lists are lists of node identifiers of arbitrary length, and where <ctrl list1> and <ctrl list2> must have the same number of control node identifiers, and <sigl list1> and <sigl list2> must also have the same number of elements.

1.2.15. PAD ORDER STATEMENT

In order to facilitate bonding and printed circuit board chip interconnection, it is possible to specify the pad order required. Since there is no default pad ordering it is mandatory to have a pad order statement within each chip definition. There are three fixed positions : VDD, GND, CLOCKS. This gives three fields within which the remaining pads can be placed. The physical design com-

piler attempts to space the pads equidistant with respect to each other, within each field.

The pad order statement has the following form.

Padorder VDD,(xxx,...,xxx,)GND,(xxx,...,xxx,)CLOCK,(xxx,...,xxx)

Where xxx is replaced by an external node identifier, and the number of xxx replacements within each field is arbitrary.

The choice of pad order will be determined by system wiring requirements on chips and on the pad order consequences on floor plan and chip area.

1.2.16. STATEMENTS

Statements are generally written on one line and are either declarations, instantiations, end statements, comments, assignment replacements or pad order statements. Where necessary a statement may be broken over more than one line, by using a continuation character(-) as the last character on the line to be continued. The continuation character may be omitted if the character preceding it is a comma.

The Continuation Character is the character -

1.2.17. PROGRAM STRUCTURE

A program consists of a set of statements; its structure requires declaration before invocation. A program should (by convention) start with some comment statements giving identification etc., which are followed by the necessary declarations and calls in sequence. No nesting or recursion is permitted. Functional element

declarations must each have their own corresponding end statement. The final line of the program must be the end of program statement (apart from subsequent comments, which are to be ignored).

1.2.18. CHECKS, DIAGNOSTICS AND WARNINGS

Because human generated coding is error prone, it has been a high priority objective to incorporate as much checking together with diagnostic warning of detectable errors as can be usefully implemented in each of the compilers and at each stage of compilation, as appropriate. The purpose of these checks and diagnostics is twofold :

1. to exclude , or at least warn of faulty constructions during each stage of implementation.

2. to aid in system design debugging.

Definitive verification of system function is given by the behavioural description compiler, but the earlier checks and warnings help to eliminate most faults as early as possible, often before simulation. The checks carried out by each compiler are detailed in the appropriate section.

The language compiler includes checks (and warns on the occurrence of error conditions) for the following :

checks on names
checks on types
checks on declarations
checks on parameters
checks on signals & controls
checks on pads
checks on connections
checks on statements
checks on syntax

There are approximately fifty kinds of check made during language compilation.

Table 3 shows the formal syntax definitions and the last section gives a worked example to illustrate each of the language constructs as used for system specification.

TABLE 3.

```
<STATEMENT>=<NAME>'='<LIST>;
        <OPTYPE><NAME><FLIST><CTLLIST><LIST><OLIST>,
        "SIGNAL"<LIST>,
        "CONTROL"<LIST>,
        "PADIN"<EQUIVC><EQUIVS>,
        "PADOUT"<EQUIVC><EQUIVS>,
        "PADORDER"<LIST>,
        "CONSTANT"<CONSTDEFS>,
        "ENDOFPROGRAM",
        "END",
        "CONTROLGENERATOR"<CTLLIST>,
        "ENDCONTROLGENERATOR",
        <NAME><PARAMLIST><CTLLIST><LIST><OLIST><REPLIST>,
<PARAMLIST>='['<PLIST>']', ;
<PLIST>=<EXPR>','<PLIST>,<EXPR>, ;
<FLIST>='['<LIST>']', ;
<CTLLIST>='('<LIST><OLIST>')', ;
<OLIST>='->'<LIST>, ;
<LIST>=<NAME>','<LIST>,<NAME>, ;
<EQUIVC>='('<LIST>'->'<LIST>')', ;
<EQUIVS>=<LIST>'->'<LIST>, ;
<REPLIST>="TIMES"<CONST>"WITH"<CTLLIST><LIST><OLIST>, ;
<CONSTDEFS>=<ASSIGN>','<CONSTDEFS>,<ASSIGN>, ;
<ASSIGN>=<NAME>'='<EXPR>;
<EXPR>=<TERM><ADDOP><EXPR>,<TERM>;
<TERM>=<FACTOR><MULTOP><TERM>,<FACTOR>;
<FACTOR>=<ADDOP><UNSIGNED>,<UNSIGNED>;
<UNSIGNED>=<NAME>,<CONST>,'('<EXPR>')';
<ADDOP>='+','-';
<MULTOP>='*','/';
<OPTYPE>="OPERATOR","CHIP","SUBSYSTEM","SYSTEM";
$
```

1.3. Upgrades and Enhancements

A number of new features have been added to the language since this version. The underscore character has been included as a legal character for names. Modifications and additions have been made to the repeated call and assignment replacement syntax, and a short hand for long lists of node names has been added These changes are outlined below.

1.3.1. Shorthand for long signal lists

Frequently the designer specifies an object which is in turn

composed of many identical objects working concurrently on a long input data list, producing a long output data list. The SUBSYSTEM Outcolumn in our example is one such object, taking in sixteen real and sixteen imaginary signals, and producing sixteen magnitude outputs. These lists can usually be named in a repetitive manner, and FIRST allows their representation in shorthand using the THROUGH statement. A nodelist of the form alphanumber, alphanumber+1, ... , alphanumber+N may be represented by alphanumber THROUGH number+N, e.g.:

sig31, sig32, sig33, sig34, sig35, sig36

may be represented by:

sig31 THROUGH 36

The first line of the declaration of Outcolumn, naming forty-nine nodes, looks like this in shorthand:

```
SUBSYSTEM OutColumn (c1) re1 THROUGH 16,
                    im1 THROUGH 16 -> mag1 THROUGH 16
```

1.3.2. Shorthand for repeated instantiations and linear arrays

Modular architectures often require repeated instantiations of identical objects. FIRST provides a syntax for condensing such repeated instantiations into one statement. This is done by appending, to a normal instantiation, a phrase which defines the repetition and connection structure. The form of this phrase is as follows:

TIMES constant WITH cascade

where cascade is a phrase whose form and function is explained below. The full form of a repeated instantiation is:

name [param list] (ctrl list) siglist TIMES constant WITH cascade
The number of repetitions is defined by the value of constant and the different types of connection are defined by cascade.

Cascade can be a null string. In this case corresponding inputs and outputs of the repeated element are connected in common to the nodes named in the instantiation. An example of this is illustrated in Figure 1 and is represented syntactically as:

```
!============================================================
OPERATOR F1 in -> out
     BITDELAY [1] in -> out TIMES 3 WITH
END
!============================================================
```

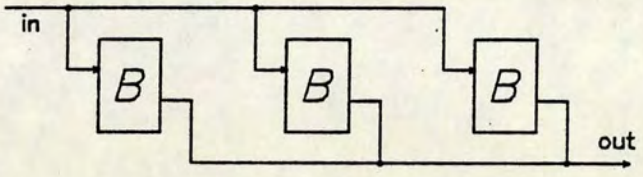


Figure 1

This syntax may describe a legal structure for inputs (though fan in may be a problem if the constant value is large) but the connection of more than one output to any node is not normally a legal construct. The mechanism for severing the corresponding output or input connections from this common node is a statement of assignment replacement, which follows the repeated instantiation. The form of this is:

name = list

where name is a node identifier (occurring in the repeated instantiation) and list is a list of the new node identifiers to be used as replacements. The effect of this statement is to replace each occurrence of name by the next name in the list, starting from the first occurrence and the first name. This substitution continues down to the assignment replacement statement or until the list is exhausted. An example of this type of connection of repeated elements is illustrated in Figure 2. The syntax for this is:

```
!================================================================
OPERATOR F2 in1, in2, in3 -> out1, out2, out3
     SIGNAL in, out
     BITDELAY [1] in -> out TIMES 3 WITH
       in = in1, in2, in3
       out = out1, out2, out3
END
!================================================================
```

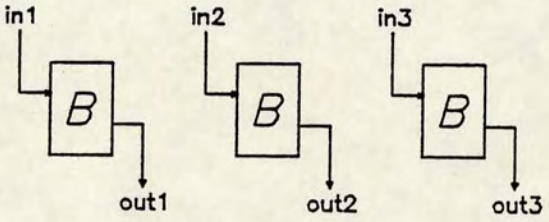


Figure 2

Thus we have a syntax for describing global and distinct connections in repeated instantiations. In order to describe other modes of connection for repeated elements it is necessary to use the cascade phrase. This allows the description of four different

types of nearest neighbour connection, one or more of which may be used. The types of connection available are:

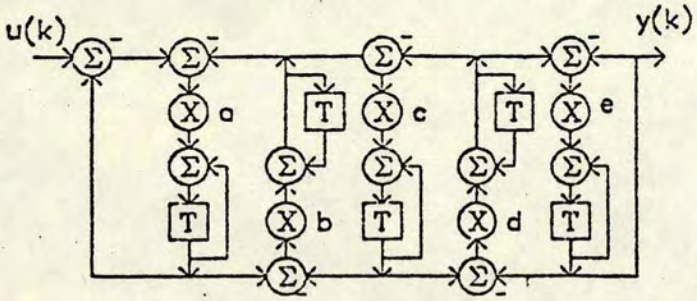
1) Forward
2) Forward tapped
3) Backward
4) Backward tapped

The phrase cascade has four components, one for specifying the connections of each type. (Some or all may be empty, according to whether or not there are connections of the particular type). Figure 3 illustrates each of these types of connection, and the corresponding syntax is as follows.

```
!================================================================
! Forward cascade
OPERATOR F3 in -> out
     BITDELAY [1] in -> out TIMES 3 WITH -
       out -> in
END

! Forward tapped cascade
OPERATOR F4 in -> tap1, tap2, tap3
     SIGNAL out
     BITDELAY [1] in -> out TIMES 3 WITH -
       out => in = tap1, tap2, tap3
END

! Backward cascade
OPERATOR F5 in -> out
     BITDELAY [1] in -> out TIMES 3 WITH -
       in <- out
END

! Backward tapped cascade
OPERATOR F6 in -> tap1, tap2, tap3
     SIGNAL out
     BITDELAY [1] in -> out TIMES 3 WITH -
       in <= out = tap1, tap2, tap3
END
!================================================================
```

Each of the cascade phrases takes the form of lists of nodes on either side of the cascade assignment symbol (->, =>, <-, or <=). In the case of the simple forward and backward cascades the internal signal nodes are not externally named. However, a new set of

Figure 1.1 5th order LDI implementation



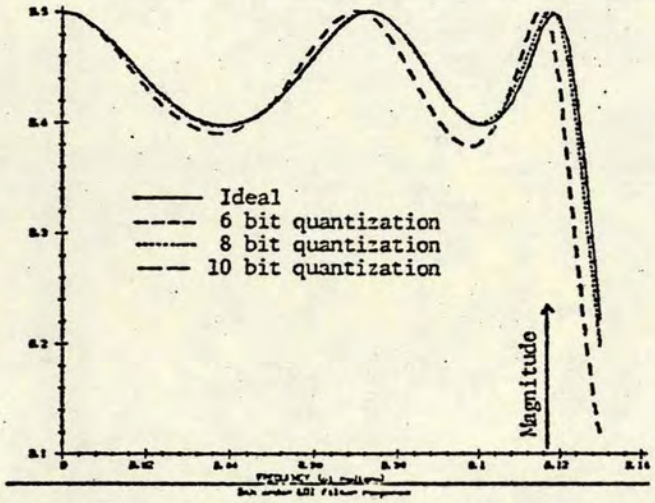


Figure 1.2 5th order LDI magnitude response

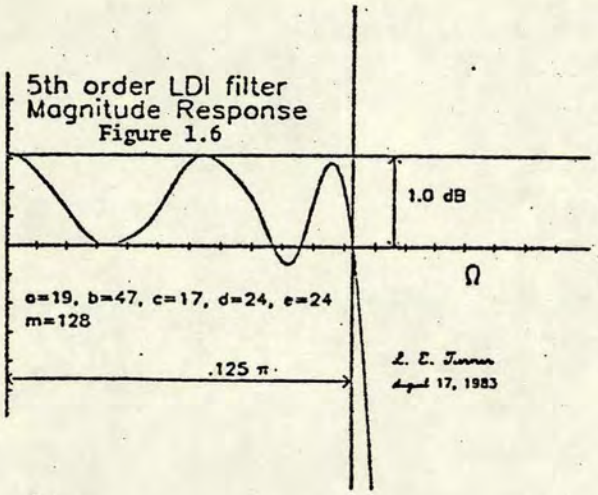
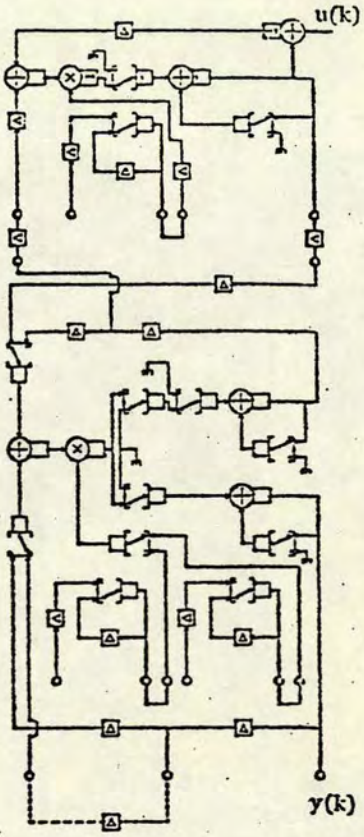


Figure 1.3 Bit serial LDI filter



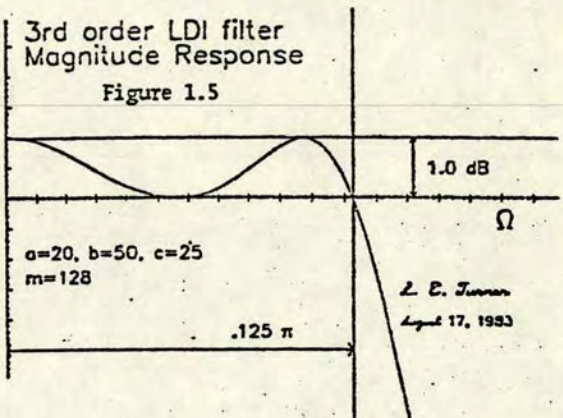
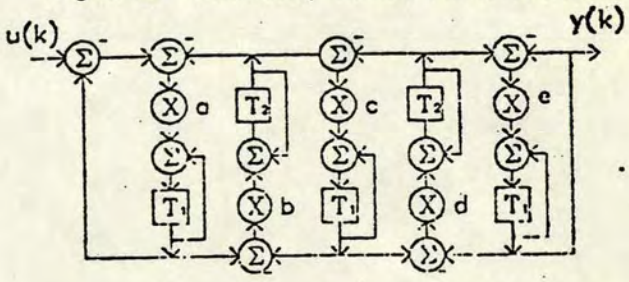


Figure 1.4 Alternately clocked 5th order LDI



1.5. Multiplier Multiplexing

An advantage of bit serial arithmetic over parallel arithmetic is obtained when arithmetic elements can be shared by using multiplexing. If the signal bandwidth of the final product without sharing arithmetic elements is greater than necessary then a silicon area saving can be realized. Since the serial multipliers are the largest single arithmetic element they are the most likely candidate for sharing.

In the case of LDI filters, it is desirable that the filter operate as fast as possible to allow a variety of applications. However, an efficient LDI digital filter implementation is possible if one multiplier is shared between stages of a second order section. This type of sharing is impractical in parallel LDI filter implementations because of the increase in the number of interconnections required.

To explain the multiplier sharing used in LDI filters consider the LDI digital filter implementation given in figure 1.4. If every second stage in the signal flowgraph in figure 1.4 is updated alternately then the same signal flowgraph as in figure 1.1 is implemented. This alternate clocking scheme has the advantage of simplifying the LDI filter signal flowgraph (regular structure) in addition to saving one multiplier for every two sections.

1.6. Initialization

The FIRST system allows an external event to be recognized by the LDI filter implementation. This external event is used for two purposes. All state registers are cleared and the filter coefficients are loaded into registers before the filter begins operation.

1.6.1. State Registers

Since the LDI digital filter is a recursive system it is necessary that the filter state registers be set to zero before the filter is allowed to operate. The state registers in the FIRST implementation of the LDI digital filter are implemented as bit delay elements associated with an adder element. To set these bit delay elements to zero multiplexers are used to open the feedback loop and set the value entering the adder to zero. Note that because of the alternate clocking scheme used, in some cases only one multiplexer is needed as the input is already zero during the initialization.

1.6.2. Filter Coefficients

The filter coefficients are loaded as serial unsigned binary values during the initialization phase. The same control signals used to set the state registers to zero are used to load the filter coefficients. Multiplexers and signal bit delays are used to recirculate the multiplier coefficient with the correct timing and synchronization.

1.7. FIRST Simulation

The FIRST simulator is event driven thus efficiently allowing both control and data signals to be simulated. This is an essential step as all control signals used by each arithmetic element must be synthesised by the device implemented.

The LDI filter was simulated by using a single large valued (100,000) input to approximate an impulse input. The resulting simulated response was transformed to the frequency domain using a fast fourier transform program. The results of the third and fifth order LDI digital filter simulations are shown in figures 1.5 and 1.6 respectively.

1.8. Conclusion

The design of a recursive digital LDI ladder filter using the silicon compiler FIRST has been described. The silicon compiler currently supports the implementation of devices using a 5 micron silicon gate nmos process. The clock rate is eight Mhz. With a system wordlength of 22 bits a sample rate of approximately 180 Khz is possible.

1.9. References

[1] Bruton, L. T., 'Low sensitivity digital ladder filters', IEEE Trans. 1975, CAS-22, pp. 168-176

[2] Denyer, P.B., Renshaw, D. and Bergmann, N., 'A silicon compiler for VLSI signal processors', proc. European Solid State Circuits Conference, Brussels, 1982.

[3] Denyer, P.B. and D. Renshaw, 'Case studies in VLSI signal processing using a silicon compiler' IEEE ICASSP 83, 1983, Boston, pp. 939-942

[4] Liu, E. S. K., L. E. Turner, and L. T. Bruton, 'Exact synthesis of LDI and LDD ladder filters' Accepted for publication by the IEEE Trans. on Circuits and Systems.

[5] H. Orchard, "Inductorless filters', Electronics Letters, vol. 2, pp. 224-225, June 1966.

[6] Vaughan-Pope, D.A. and L.T. Bruton, "Transfer function synthesis using generalized doubly terminated two-pair networks', IEEE Tran. on Circuits and Systems, vol. CAS-24, no. 2, pp. 79-88, February 1977.

[7] Bruton, L.T. and R.H. Khan, ' Multirate multiplierless discrete ladder filters', Electronics Letters, vol. 14, pp. 814-815, 1978.

[8] Lyon, R.F., ' A bit-serial VLSI Architectural Methodology for signal processing', VLSI 81, University of Edinburgh, August 1981, pp. 131-140.

P. B. Denyer and D. Renshaw are with the Integrated Systems Group, Department of Electrical Engineering, University of Edinburgh, The King's Buildings, Edinburgh, UK.

L.E. Turner is with the Department of Electrical Engineering, University of Calgary, Calgary, Alberta, Canada.

Removing of the unrealizable advance operators in the filter terminations does not eliminate the desirable sensitivity characteristics of this filter. However, the digital filter can no longer be designed exactly by direct transformation from an analog prototype filter. (If the values of the inductor and capacitor elements in the analog filter are used to define the multiplier coefficients in the digital filter errors will occur in the transfer function realized.) Exact methods of synthesis are available for the LDI lowpass ladder filter[6] including a complete design program for LDI filters[4]. In addition, optimization methods have been used to design LDI filters[7].

1.3. Finite Precision Effects

In any practical recursive digital filter implementation, signal levels are represented by finite, quantized binary values. The quantized binary values are also used to represent the values of the filter multiplier coefficients. The discrete approximation of a continuous integrator is actually implemented as a digital integrator. Any digital implementation is limited in accuracy by the finite quantized nature of the digital signals on which it operates. The binary filter multiplier coefficient value is a quantized approximation of the ideal filter multiplier coefficient value. This quantization causes a deviation in the actual filter transfer function as compared to the ideal filter transfer function.

Nonlinearities associated with adder overflow and signal level quantization occur in any implementation of a recursive digital filter. These nonlinear operations often lead to undesirable autonomous oscillations or limit cycles.

1.3.1. Coefficient Quantization

The effects of coefficient quantization in an LDI digital filter implementation are less severe than in other digital filter structures[1]. This is due to the low sensitivity property of the analog prototype filter. The ideal magnitude response of a fifth order LDI digital filter is given in figure 1.2. For comparison, in the same figure, the magnitude response obtained using six, eight and ten bit coefficient quantization is given.

1.3.2. Signal Wordlength

If the required signal value at any node in the LDI filter becomes larger than the binary wordlength available, then an overflow occurs.

Overflow causes very severe signal distortion in the filter. In any practical filter implementation it is necessary to ensure that under expected operating conditions, overflow does not occur. This is done in most filters by scaling the input signal to the filter such that the gain to any internal node is less than unity. In LDI filters it can be shown that scaling is not required since the gain from the input to any internal node is always less than unity. This assumes that the signal paths in the digital integrator are larger than the input signal wordlength by an amount equal to the coefficient quantization.

1.3.3. Limit Cycle Oscillations

Limit cycle oscillations are undesirable nonlinear effects which are due to the finite quantized nature of the digital filter implementation. Two types of limit cycle oscillations are recognized. These are overflow oscillations and granularity oscillations. Providing that the the LDI filter terminations are implemented with infinite precision (unity gain), then no serious limit cycle oscillations are known to occur.

1.4. FIRST Implementation

The signal flowgraph of the bit serial LDI digital filter implemented using FIRST is shown in figure 1.3. This is a third order LDI digital filter organized as a first order input section and a separate second order section. This filter is designed so that fifth, seventh or higher odd order filters can be implemented by cascading the second order sections. Using the FIRST compiler has several advantages. Multiplexers can be efficiently used to share arithmetic elements because of the use of bit serial arithmetic. Both the system wordlength and the multiplier coefficient wordlength can be parameterized to allow easy modification for different applications or testing. The user does not need to know the fine detail of the arithmetic element implementation.

1.4.1. Pipelined Serial Arithmetic

The FIRST compiler generates an integrated circuit design with arithmethic elements which use bit serial pipelined arithmetic[8]. The operations of multiplication, addition, subtraction, multiplexing, and bit delay are used in this filter implementation. Each aritmentic element has a known latency. Latency is the time required (in terms of bit delays) for each arithmetic element to complete its operation. Because of this pipelining each FIRST element also requires control signals which identify when the least significant bit of the data signals will arrive.

The use of pipelined serial arithmetic allows all filter operations to be performed as soon as data from previous elements is available. The recursive nature of the LDI filter places a restriction on the latency or delay around any feedback loop. That is, the latency around any feedback loop must be equal to the system wordlength. This requires that additional bit delays be incorporated into both the signal and control paths.

1.4.2. Parameterization

The FIRST compiler allows parameters to be included in the filter description. These parameters appear as named variables. The system wordlength and the coefficient wordlengths have been parameterized in the LDI filter design. This allows filters with different wordlengths to be easily implemented and tested. The restriction that any feedback loop have a latency equal to the system wordlength is included.

A BIT SERIAL LDI RECURSIVE DIGITAL FILTER

L.E. Turner+, P.B. Denyer* and D. Renshaw*

*Integrated Systems Group
Dept. of Electrical Engineering
University of Edinburgh, U.K.

+Dept. of Electrical Engineering
The University of Calgary
Calgary, Canada

Abstract

The practical implementation of a bit serial lossless discrete integrator (LDI)[1] recursive ladder filter suitable for implementation as a single integrated circuit is described. The low coefficient sensitivity and simplicity of the LDI signal flowgraph make the filter structure suitable for implementing high quality digital filters. A bit serial integrated circuit filter has been designed and simulated using the silicon compiler system FIRST[2,3].

The LDI filter coefficients which yield a Chebyscheff transfer function characteristic are found using an exact synthesis method[4]. The finite precision time domain response of the filter is simulated using the FIRST simulator and the magnitude response is verified by calculating the fourier transform of the filter unit sample response. The LDI filter implementation makes use of an alternate clocking scheme which simplifies the signal flow graph. This is a form of multiplexing which is easily implemented using bit serial arithmetic.

1. LDI Recursive Ladder Filter Design and Implementation

1.1. Introduction

The implementation of a high order recursive LDI digital filter[1] as a single integrated circuit using parallel arithmetic is unlikely using currently available integrated circuit technologies. Digital LDI lowpass ladder filters have two input signal paths and two output signal paths for each integrator stage. The large number of parallel internal interconnections required makes very poor use of the integrated circuit area. Each integrator section requires a parallel multiplier which occupies the majority of the chip area. These constraints make the integration of a high order parallel architecture single chip LDI ladder filter impractical. Integrated LDI digital ladder filter implementations have been constructed. However, these implementations use single chip parallel integrator sections with external connections and avoid the need for multipliers by using a multirate clocking scheme.

Bit serial arithmetic implementations of recursive digital filters do not have this interconnection problem. In addition, the serial arithmetic elements are much smaller than the equivalent parallel elements. Fifth or higher order single chip recursive LDI ladder filters are possible using bit serial arithmetic.

The LDI lowpass ladder filter is known to exhibit low transfer function sensitivity to the filter coefficients[1]. This is due to the excellent low sensitivity characteristics of the resistively terminated analog ladder filter from which it is designed. (The analog filter is almost insensitive to variations in the inductor and capacitor values providing that the passband insertion loss is almost zero[5]).

This section describes the implementation of a bit serial recursive LDI ladder filter using the silicon compiler system FIRST[2,3]. A single chip integrated circuit LDI digital filter is design and simulated using FIRST.

1.2. LDI Ladder Filter Implementation

The flowgraph of a digital LDI ladder filter is obtained from the voltage and current (VI) equations of a resistively terminated inductor, capacitor ladder filter.

To implement a digital ladder filter, each continuous integrator element is replaced with a discrete approximation. This is equivalent to applying the transformation

$$s=(z-1)/(z^{-1}) \quad (1)$$

to the Laplace transform transfer function of the analog filter (H(s)). Replacing z with -1/z in the transformation above yields the same transformation. Therefore a digital filter designed using this transformation will have transfer function poles both inside and outside the unit circle. This difficulty can be overcome by scaling the transfer function. Algebraic manipulation of the transfer function or flowgraph manipulation of the signal flowgraph equivalent to impedance scaling the network by exp (-sT/2) yields the scaled LDI given in figure 1.1. The unrealizable advance operators have been removed to ensure that the resulting transfer function is stable.

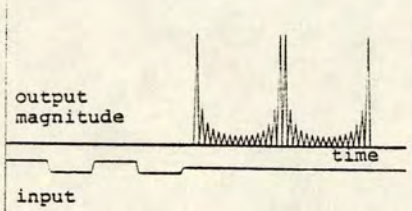


Figure 3a. Simulated DFT with 32 modules (N=32). Rectangular input waveform and the output has been converted to magnitude.

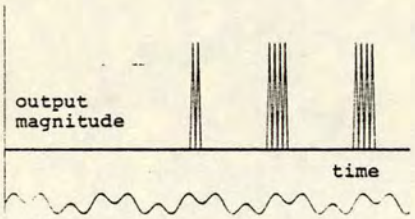


Figure 3b. Simulated DFT with 32 modules for an input of two tones: fundamental and third harmonic.

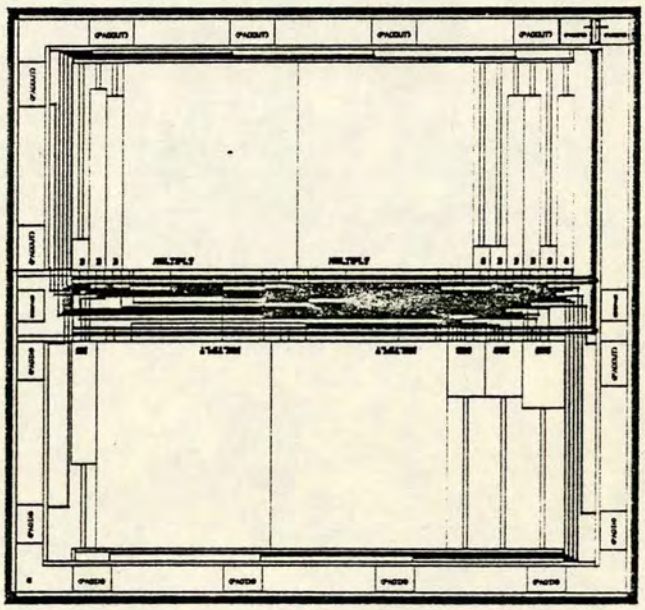


Figure 4a. Chip floorplan produced by the FIRST Silicon compiler for the DFT module with coefficient length of 16 bits and wordlength of 28 bits with complex multiply having four real multiplies.

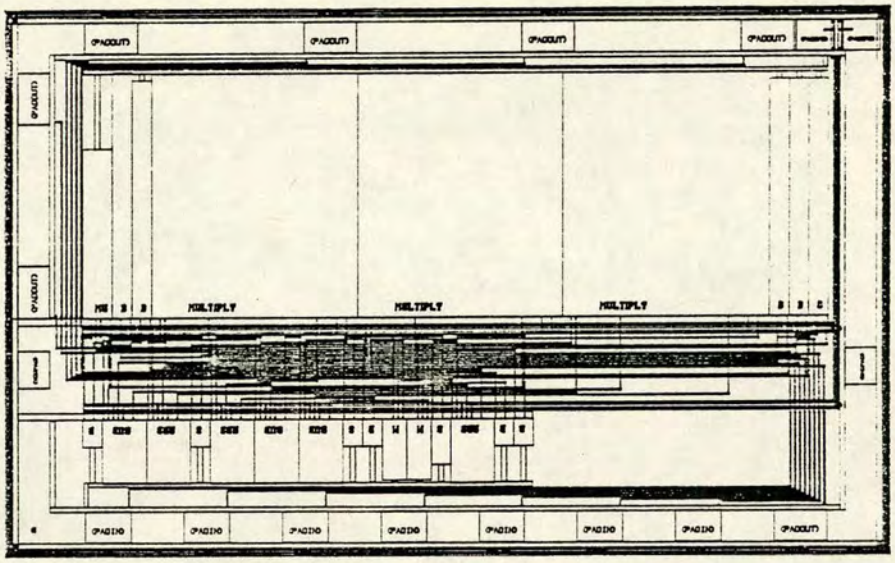


Figure 4b. DFT chip floorplan for coefficient length of 16 bits and wordlength 30 bits for the complex multiply performed using three real multiplies. Figures 4a and 4b are not to the same scale. This chip has smaller area than the chip in Figure 4a.

point DFT with input being two cycles of a rectangular waveform. This figure shows the delay through the array between arrival of data and production of the DFT result.

The floorplan of the chip for this Fourier transform module is shown in Figure 4. By time multiplexing the arithmetic hardware in the modules using multiplexors and additional storage, the number of chips required could be reduced by the multiplexing factor for applications which could afford the reduced rate of data throughput.

CONCLUSION

This paper has demonstrated the use of a linear systolic array as a discrete Fourier transform engine. The number of processor modules required is proportional to the number of points in the transform, N. The delay between acceptance of data and production of the output and the data storage required are also proportional to N. The interconnection of modules is simple with data and the coefficients W supplied at one end of the array and the transform result available at the other. For simplicity of interconnection and storage of input data and intermediate results, this scheme is superior to the area-time efficient algorithms discussed by Thompson [2].

ACKNOWLEDGEMENTS Part of the work reported here was performed while one of the authors (G.H.A.) was a Visiting Scientist at the Research Laboratory of Electronics at the Massachusetts Institute of Technology. The FIRST compilation and simulation was performed in the Department of Electrical Engineering at the University of Edinburgh. The FIRST compiler has been developed under grants to the University of Edinburgh from the U.K. Science Research Council.

REFERENCES

[1] H.T. Kung, "Special purpose devices for signal and image processing: an opportunity in very large scale integration (VLSI)", SPIE Vol. 241, Real Time Signal Processing III, pp. 74-84, 1980.

[2] C.D. Thompson, "Fourier transforms in VLSI", IEEE 1980 Conf. on Circuits and Computers, ed. N.B. Guy Rabbat, Pt Chester, N.Y., Oct 1-3, pp. 1046-1051, 1980.

[3] P.B. Denyer, D. Renshaw, and N.W. Bergman, "A silicon compiler for VLSI signal processors", ESSCIRC '82 Digest of Technical Papers, pp. 215-218, 1982.

[4] P.B. Denyer and D. Renshaw, "Case studies in VLSI signal processing using a silicon compiler", ICASSP 83, IEEE International Conference on Acoustics, Speech and Signal Processing, Paper 20.6, pp. 939-942, 1983.

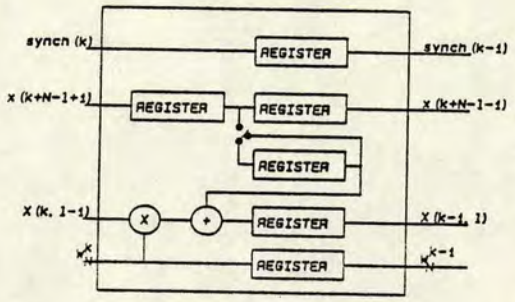


Figure 1. The basic DFT module. Latching of coefficient data is controlled by the synchronizing signal.

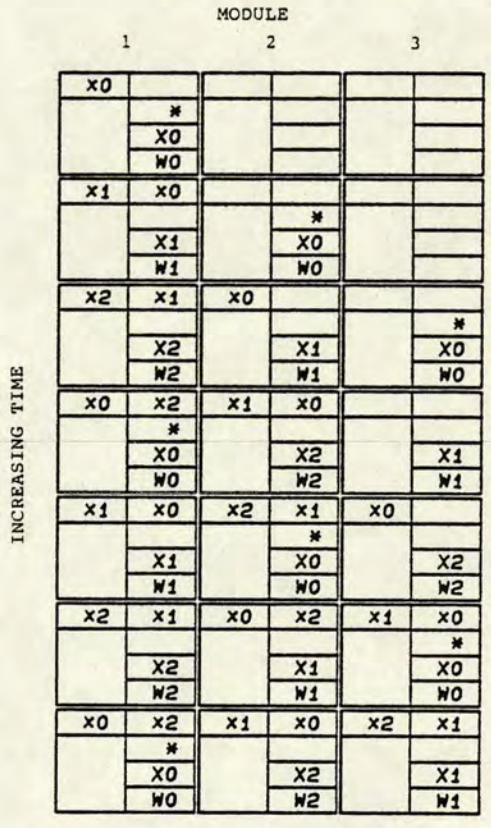


Figure 2. Example of the linear array DFT for an N=3 point transform. The partial result entering at the left is zero and the transform accumulates as these X0, X1 and X2 move through the array.

in reverse order. This can be arranged by passing the data through the array and latching coefficient data successively in each module.

The function of each of the modules in the array is described by Figure 1. The partial result, the W factor and the synchronizing signal move through the array with the same delay in each module. This delay time is the time to perform the complex multiply and add operations. This is referred to here as the *module time*. The timing requires that each coefficient be delayed an extra module time in each module. In this way x(N-1) is latched into the first module one module time before x(N-2) is latched into the second module and so on until x(0) is latched into the last module.

The timing diagram for a three point (N=3) DFT (Figure 2) illustrates the flow of data and results through the array. The data, W factors and partial results contained in registers in each of the modules (across the diagram) are shown at successive times (down the diagram). The instant of latching of data into the appropriate register in the modules is indicated by an asterisk. Data are shown moving through the array while the previous batch of DFT values are being computed. Following the arrival of the last of the datum of a batch, x(N-1), into the first module, the computation of the DFT for that batch of data proceeds while data of the next batch move through the array. The first output for a batch, X(0), is obtained N module times after the arrival of the last datum for the batch, x(N-1), and this is 2N module times after the arrival of the first datum x(0). The last output for the batch, X(N-1), is available N-1 module times after X(0).

The linear array has the useful feature that the data are stored within the array. This makes direct comparison in an area-time efficiency sense with the algorithms described by Thompson [2] difficult since with this algorithm data storage is included whereas with most of the algorithms considered by Thompson data storage is not included.

The linear array also has the desirable feature that data enter the array at the same rate as the results are produced. This means that the transform system can be interposed between a data sampling system and the recording system to produce a discrete Fourier transform of the data, the only penalty being a delay equal to twice the transform length N.

The development to this stage does not depend on the nature of the arithmetic to be used. Since bit serial arithmetic for this array leads to fewer interconnections between modules than for a bit parallel implementation and since it is possible to design the complex multiply and add operations for bit serial arithmetic on a single chip for 5 or 6 micron VLSI, the algorithm was implimented in serial arithmetic using the *FIRST* silicon compiler [3].

The module consists mainly of a complex multiply and accumulate operation. While it is possible to configure the complex multiply using only three real multiplies, such a scheme leads to a longer wordlength in both the serial and parallel implementations because of the extra addition and subtractions which become necessary. The saving in the use of three multipliers instead of four is partly lost in the larger multipliers required for the larger wordlength in addition to a time penalty for the arithmetic operations.

IMPLEMENTATION AND SIMULATION USING FIRST

The FIRST silicon compiler is a system of computer programs which enables a designer to specify a signal processing task as a set of primitive operations, simulate the task and obtain a 6 micron NMOS VLSI layout of the chips in the system [4]. For the linear array DFT modules the primitives used were serial multiply, add, subtract and delay together with a specification of the control signals.

For example, the description of the complex multiply is as follows:

```
OPERATOR XMULT [coeff] (cin -> cout) rd,id,rc,ic
                                     -> rout,iout

SIGNAL r1, r2, i1, i2
CONTROL c1
MULTIPLY [1,coeff,0,0] (cin -> c1) rd, rc -> r1,NC
MULTIPLY [1,coeff,0,0] (cin -> NC) id, ic -> r2,NC
MULTIPLY [1,coeff,0,0] (cin -> NC) rd, ic -> i1,NC
MULTIPLY [1,coeff,0,0] (cin -> NC) id, rc -> i2,NC
SUBTRACT [1,0,0,0]     (c1)  r1, r2 GND -> rout,NC
ADD      [1,0,0,0]     (c1   i1, i2 GND -> iout,NC
CBITDELAY [1]          (c1 -> cout)

END
```

where the parameters in round brackets () are control lines, the parameters in square brackets [] are functional parameters and signal lines are rd, id, rc and ic representing the real and imaginary parts of the multiplier and coefficient, and rout and iout representing the real and imaginary parts of the product.

By use of control primitives to effect multiplexing of data lines, bit delays and specification of connections to pads of the chip, the chip representing the module in Figure 1 was described in approximately fifty lines of code similar to that given above. For the simulation, the modules were cascaded so that the output resulting from sequences of input data could be obtained. A typical output is shown in Figure 3 which is the magnitude of the simulated thirty-two

A BIT SERIAL LINEAR ARRAY DFT

Gregory H. Allen[+], Peter B. Denyer* and David Renshaw*

[+]Department of Electrical and Electronic Engineering,
James Cook University of North Queensland, Australia 4811.

*Department of Electrical Engineering, University of Edinburgh,
The King's Buildings, Mayfield Road, Edinburgh, EH9 3JL.

ABSTRACT

A linear array which computes the DFT in a pipelined fashion is described. The algorithm is derived from the batch processing array proposed by H.T. Kung [1] but has been modified to allow continuous operation. This computation of the DFT is a complex polynomial evaluation on the unit circle using Horner's method having the data for the polynomial coefficients. Data and the N-th complex roots of unity are input at one end of the array and the DFT sequence is output from the other. The polynomial coefficients are stored in successive modules in the array and a new batch is latched successively with a synchronising signal. In its simplest form the design has a single system part which is replicated N times for an N-point transform. For time multiplexed modules the system throughput and hardware can be optimised for given applications. A bit serial layout for 6 micron NMOS VLSI has been designed and simulated using the FIRST silicon compiler at the University of Edinburgh.

INTRODUCTION

Fourier transform hardware is widely used in signal processing. From area-time considerations, the fast Fourier transform algorithms have enjoyed a major role (see for example Thompson [2]), especially with sequential processors. This paper examines a different point of view and considers a direct implementation of the discrete Fourier transform (DFT) which is not area efficient but has very simple implementation using a cascade of identical parts. This work is an extension of the systolic array DFT structure of Kung [1] which included a loading phase and a computational phase. The extension is the prescription of a system which loads and computes simultaneously. The data are input at one end of the array and delayed Fourier transformed data emerge from the other end in natural order.

THE DISCRETE FOURIER TRANSFORM ALGORITHM

The N point DFT of a data sequence x(n), n=0,1,2, ...,N-1 is computed from the equation (1)

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad (1)$$

for k=0,1,2, ...,N-1 with $W_N = \exp(-j2\pi/N)$.
This equation can be rearranged using Horner's method to obtain

$$X(k) = ((...((x(N-1))W_N^k + x(N-2))W_N^k + ..$$

$$.. + x(1))W_N^k + x(0)) \qquad (2)$$

Kung [1] has shown how to perform this computation using a linear systolic array of processors. The DFT values X(k) are computed sequentially by passing the kth partial result together with the complex value W_N^k into a computational module containing the coefficient x(j), multiplying the kth partial result by W_N^k and adding x(j) to produce the next kth partial result. When the initial kth partial result entering the array is set at zero and the complex values W_N^k are supplied in the sequence $W_N^0, W_N^1, W_N^2, \ldots W_N^{N-1}$ the result emerging from the linear array will be X(0), X(1), ..X(N-1) provided that the coefficients x(j) in the modules are in the order x(N-1), x(N-2), x(N-3), ... , x(1), x(0).

The data entering the linear array are provided x(0) first, x(1) next, and so on until x(N-1) arrives. However the data coefficients to be stored starting with the first module should be

In the TTL hardware word growth was accommodated by hard limiting in the arithmetic unit. It was not feasible to incorporate this at each calculation in the integrated design, and as the likely word growth in the lattice was not known in advance, it was assumed that the signals were of gaussian distribution and room was provided accordingly for word growth.

The probabilities of an overflow occurring, and thus the room allowed for word growth, were set according to the penalties which overflow would incur. Too great an input signal sample voltage would be clipped by the input ADC, causing signal distortion. The probability for this was set to 1 in $\sim$ 100. The possibility of overflow which causes wrap-around within the filter signal path and results in a burst of errors during equalisation, was held below 1 in 10^7. As the system can not recover from a wrap-around in the modified stepsize recursion, it was prevented completely using limiters.

A comparison with the CORDIC integrated lattice filter design (8) developed for speech analysis, which is based on 16 bit coordinate rotation arithmetic shows that the latter uses an area smaller than our chip set. The CORDIC normalised lattice approach has a different arithmetic capability to our FIRST approach, avoiding the need for stepsize recursion.

In order to realistically apply CORDIC techniques, even to 8 kHz sample rate signals, one has to employ parallel processing (8) to minimise the internal clock rate and permit multiplexing. A 12 stage vocoder was reported (8) as requiring a two chip lattice cascade, operating at an 11.4 MHz internal clock rate, plus further external control circuits. In comparison our 5 chip bit serial approach (9) offers 16 lattice stages with enhanced 22 kHz sample rate for a conservative 8 MHz internal clock rate. Progress to the reduced feature dimensions of VLSI will reduce the overall chip count in both these approaches.

Application as Adaptive Equaliser

A study was also made of extending our integrated lattice filter design to the gradient equaliser of (5). This would require the addition of a side-tap structure and a recursion to set the tap weight values. Since the G weight stepsize is normally double the K weight stepsize, the same recursion would suffice for both, resulting in a requirement for only two additional chips. However decision feedback would provide a serious bottleneck for such a highly concurrent system. The error signal for the first stage can only be calculated after the output from the final stage has been obtained. However the succeeding input sample cannot be processed until the new tap value is calculated from the previous error. This problem may be solved by allowing idle cycles in the structure or by updating the taps one sample later, but neither of these solutions is particularly elegant.

A comparison was also made with the exact least squares lattice reported in (9). The extra operations include: a second PARCOR coefficient per stage; an extra multiplication and division per PARCOR recursion; an extra lattice structure for calculation of PARCOR normalising factors (ε); an extra recursion for the log likelihood variable. Using the rules developed for the prediction-error structure, this would require an additional 4-5 chips, raising the chip count for an exact least squares equaliser to about 10 chips, if implemented with our 5 μm NMOS LSI bit serial arithmetic approach.

CONCLUSIONS

We have discussed the lattice filter as one of many orthogonalising transforms used to maintain a constant rate of convergence for a linear combiner structure. We have discussed the relative advantages of lattice and transversal adaptive filters, showing that the gradient lattice is by no means ideal for all applications. Prototype TTL hardware has been described, which was constructed to investigate aspects of fixed μ gradient lattice implementation. A lattice prediction error filter chip set has been designed and compared with the CORDIC approach and the former's development into variable μ gradient and exact least squares lattice equalisers have been discussed.

ACKNOWLEDGEMENT

The sponsorship of the British Science and Engineering Research Council is gratefully acknowledged.

REFERENCES

1. Widrow, B. et al Proc. IEEE, 1975, 63, pp.1692-1716.
2. Gersho, A. BSTJ, Vol.48, Jan. 1969, pp.55-70.
3. Reed, F.A. et al IEEE Trans. ASSP-29, No.3, pp.770-775, June 1981.
4. Griffiths, L.J. ICASSP 1978, pp.87-90.
5. Satorius, E.H. and Alexander, S.T. IEEE Trans. COM-27, No. 6, June 1979, pp.899-905.
6. Ungerboeck, G. IBM J. Res. Dev., 16, No. 6, Nov. 1972, pp.546-555.
7. Denyer, P.B. et al "Case Studies in VLSI Signal Processing Using a Silicon Compiler" ICASSP 83.
8. Ahmed, H.M. et al ICASSP 1981, pp.649-653.
9. Satorius, E.H. and Pack, J.D. IEEE Trans. COM-29 No.2, pp.136-142, February 1981.

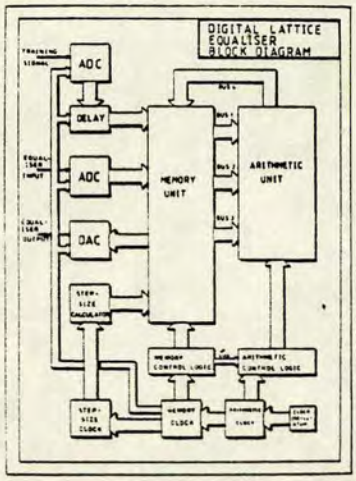


Figure 3. Discrete TTL GAL equaliser schematic.

With an arithmetic unit cycle-time of 2.2 MHz, eight operations per stage and sixteen stages, the signal sample rate is 17 kHz providing a usable bandwidth of 5-8 kHz. Approximately 160 IC's are incorporated in the equaliser. The component count was not minimised as the aim was for flexibility rather than economy.

Figure 4 shows the convergence of the 16 stage equaliser onto a real raised cosine channel whose impulse response is $0.27 + z^{-1} + 0.27\ z^{-2}$ providing an eigenvalue ratio of 11 (5). The data is convolved with this channel and then input to the equaliser 's' input, figure 1, while the 'd' port receives a synchronised undistorted binary training signal. The 'y' output shows the emergence of the equalised signal as the filter converges, with adaption started ½ ms after the start of the trace. The error output shows this reducing as convergence is achieved. The 4 ms convergence time of this fixed μ GAL equaliser is similar to that achieved in an adaptive transversal equaliser. Faster convergence is obtained by incorporating the variable μ calculation for each stage (5) as described later under the applications of the following integrated GAL PE filter design.

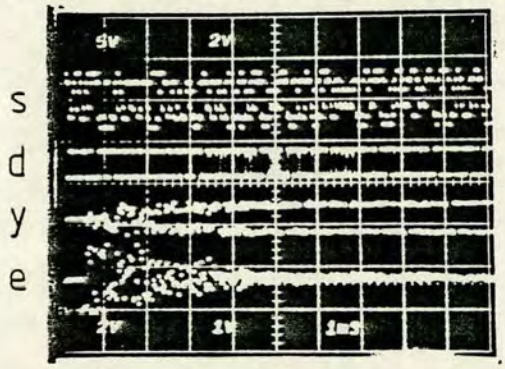


Figure 4. Shows adaption of GAL equaliser on a synthetic channel.

GRADIENT LATTICE CHIP SET

We have designed an adaptive stepsize GAL PE filter based on a set of LSI chips implemented with the fast implementation of real time signal transform (FIRST) silicon complier (7). This is a set of software tools based on bit-serial arithmetic which permitted the rapid realisation and checking of an integrated lattice filter design. The filter was partitioned into 5 distinct chip designs: lattice stage, PARCOR recursion, stepsize (μ) recursion (2 chips) and input and output multiplexing for a 16 stage filter. Figure 5 shows the floor plan of the lattice chip, which at 25 mm^2 was the largest permissible chip size with this 5 μm feature size NMOS process.

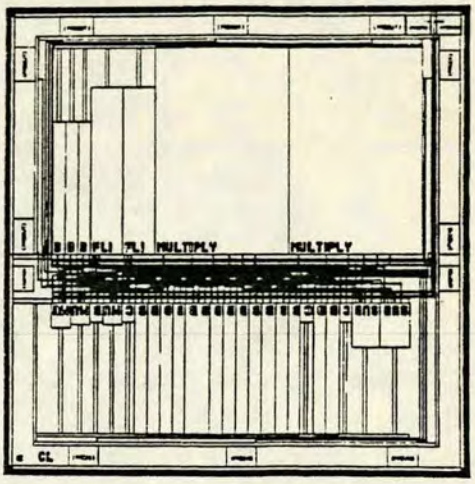


Figure 5. Floor plan of lattice stage implemented as a 5 mm x 5 mm NMOS integrated circuit with 5 μm feature size.

The stepsize recursion (5), if implemented directly, would have required 1 chip of similar area to the PARCOR recursion followed by a second operation which implemented the inversion. The FIRST system does not include the division operation necessary for the inversion. As a result the stepsize recursion itself was inverted using a Taylor expansion of the form:

$$C_n(t) = f_n(t)^2 + b_n(t)^2 \qquad (5)$$

$$\mu_n(t+1) = \mu_n(t) \cdot (1 + A - \mu_n(t).C_n(t)) \qquad (6)$$

where $C_n(t)$ is the channel power estimate and A is a scaling constant. This took only ⅓ of the total 50 sq mm maximum area permitted for the pair of chips.

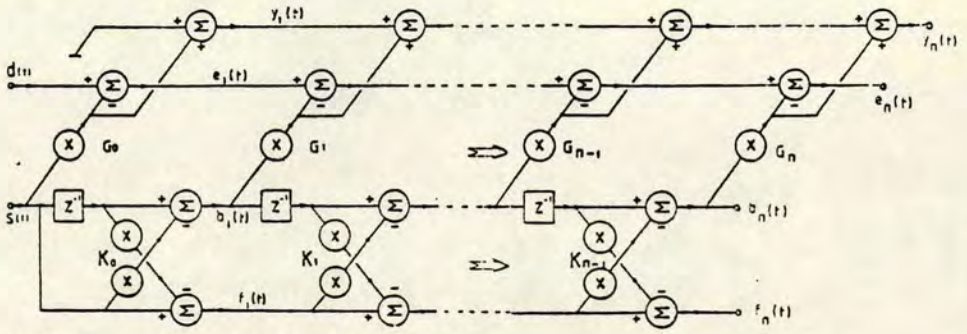


Figure 1. Adaptive lattice equaliser.

adds appreciable algorithm noise. Figure 2 shows equaliser error power (MSE) for various convergence factors. As the stepsize becomes smaller, both lattice and transversal structures approach the Weiner solution. However, as the stepsize is increased to gain faster convergence, the lattice algorithm noise grows more rapidly. Thus although the lattice has more reliable convergence properties than the transversal equaliser, it is unsuitable for applications requiring long equalisers or low converged errors.

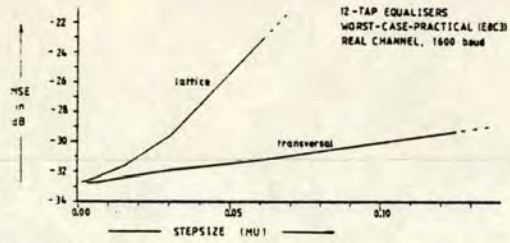


Figure 2. Converged error-vs-stepsize for constant μ lattice and transversal equalisers.

Ungerboeck (6) has shown that the error-signal power level (MSE) of an LMS algorithm in the steady state is a function of the Weiner error (e^2), the number of taps (N), the stepsize and the signal input power level (S^2).

$$MSE = 2E \langle e^2 \rangle / (2 - \mu.N.E \langle S^2 \rangle) \qquad (4)$$

where E $\langle \cdot \rangle$ is the expectation operator.

In transversal filters, where the optimum error is low, the algorithm noise is held to a minimum. In the lattice orthogonalising structure, however, the prediction error power output from each stage is dependent on the spectral characteristics of the input signal. For very low eigenvalue ratios it hardly drops at all. This leads to large fluctuations in the PARCOR values, which in turn leads to a high algorithm noise on the output of the stage. The algorithm noise is then summed as it passes through successive stages in the PE filter. For the complex equalisation which is required in a high-speed modem, the increase in the degrees of freedom in the adaptation algorithm also raises the noise by ∿ 3 dB per channel.

Our conclusions on the relative applicability of adaptive lattice and transversal filters are as follows:
If the channel eigenvalue ratio is unpredictable and reliable rate of convergence is required then lattice is the preferred approach. However lattice filters can only be applied in short equalisers where the algorithm noise is low or in applications where the background noise is so high that it masks the filter noise. The equalisation of telephone channels, where the input is often noisy, appears to be appropriate to either technique, and the final selection is dependent on the precise system specification.

TTL PROTOTYPE EQUALISER

We have developed prototype hardware to demonstrate the operation of a fixed μ gradient lattice filter with real signal inputs. This is based on a 12-bit TRW multiplier, MPY-12 HJ. The hardware was designed by partitioning the algorithm into eight multiply-and-add or multiply-and-subtract operations per stage. Twelve-bit arithmetic was selected as this represented the maximum practical working accuracy, commensurate with component availability. 24-bit accuracy was selected for the coefficient weight storage to increase recursion accuracy.

The arithmetic unit, figure 3, was designed as a multiply-and-add structure with hard limiting on overflow. Four twelve-bit busses are used for maximum throughput - three input and one output with two of them extended to 24 bits during adaptive recursion. The memory unit was designed to interface with the arithmetic unit via the bus structure. The backward-channel signal samples and filter coefficients which had to be retained and operated on repeatedly were stored in Schottky RAM. Other variables, such as the error and forward signal samples, which rippled through the filter, were kept in latching registers.

"DESIGN AND REALISATION OF ADAPTIVE LATTICE FILTERS"

M.J. Rutter, P.M. Grant, D. Renshaw and P.B. Denyer

Department of Electrical Engineering
University of Edinburgh
King's Buildings
Edinburgh

ABSTRACT

This paper compares the relative advantages of discrete and integrated circuit transversal and lattice adaptive filter designs. It discusses the trade-offs between the traditional transversal filter and the gradient adaptive lattice (GAL), showing that the reliable rate of convergence of the lattice is offset by a greater complexity and algorithm noise.

Two 16-stage hardware implementations are described. One is a TTL GAL equaliser based on a single 12-bit parallel multiplier. The second is a suite of 5 custom NMOS bit-serial arithmetic LSI chips, which together make a lattice prediction-error filter. Both implementations offer 12-bit precisions and bandwidths of over 8 kHz. This shows that the increased complexity of the lattice approach can easily be accommodated in custom VLSI circuit designs.

INTRODUCTION

Data modems are used to transfer data at high speeds over analog telephone channels of limited bandwidth. The precise frequency response of the channel is seldom known in advance of the connection and hence equalisation is needed before data can be accurately decoded. Equalisation is initially performed prior to data transmission by sending a pre-arranged training sequence, and adjusting the equaliser tap weights to minimise intersymbol interference. Data is then transmitted and the equaliser operates in a decision directed mode. Thus minimisation of the training period with a rapidly converging equaliser increases modem efficiency or throughput rate.

The most common method of realising an adaptive equaliser is to use a finite impulse response (FIR) transversal filter, where the tap-weights are calculated by the least mean squares (LMS) adaptive algorithm (1). Unfortunately, the convergence rate of the transversal equaliser is restricted by tap-weight interactions which arise from the correlation of signal components, and channels in most need of equalisation are the slowest to converge (2).

Various transforms, such as Fourier (3) and Walsh, may be applied to the input signal in order to reduce the correlation between components and thus maintain equaliser convergence rate. This paper discusses the use of the gradient lattice structure (5) as an alternative orthogonalising adaptive equaliser and demonstrates lattice filter hardware and a VLSI chip set designed for this application.

LATTICE EQUALISER

The lattice structure (4), shown in the lower part of figure 1, is a decorrelating transform, based on a family of prediction error (PE) filters. A PE filter alone may be used to equalise a signal with only pre or post echoes. However, we exploit the property of the PE filter that the backward channel outputs, b(t), are mutually orthogonal, and connect these outputs to a linear combining structure to implement an adaptive lattice equaliser (5). This alone would speed up convergence if implemented with an LMS algorithm to select the combiner weights. However, the mutual orthogonality of the signals may be further used to distribute the algorithm into many one-tap adaptive filters. This results in an economical distributed orthogonalising structure feeding a distributed linear combiner, figure 1.

The LMS adaptive equations which control the lattice filter PARCOR, values, $K_n(t)$, and combiner weights, $G_n(t)$, for stage n at time instant t are:

$$G_n(o) = K_n(o) = o \qquad (1)$$

$$G_n(t+1) = G_n(t) + 2\mu.e_n(t).b_n(t) \qquad (2)$$

$$K_n(t+1) = K_n(t) + \mu.(f_n(t).b_{n+1}(t) + b_n(t-1).f_{n+1}(t)) \qquad (3)$$

The step size μ may be fixed, or varied optimally according to the selected algorithm (5).

APPLICATIONS OF THE GRADIENT LATTICE

In the GAL filter the prediction error filters in the lattice structure must converge first to provide the orthogonal output samples which feed the distributed linear combiner. This requirement for a fast converging PE filter structure

Paper No. 205 prepared for IEEE ICASSP Boston April 1983.

Figure 10: A multichip system in 'run' mode.

diagnostic accuracy determined only by the ability to decode different signatures. If a small increase in chip complexity can be tolerated, each chip can be capable of recognising 'good-chip', 'good subsystem', 'good system' (etc) signatures, whereupon the location of a fault can be accurately pinpointed.

The chip test is essentially an off line test involving an explicit test phase of 0.25 mS. So short is the test time, however, that it may be possible in certain applications to perform a pseudo-online self-test by 'cycle-stealing'.

We have also created a regime within which fault-tolerant systems may be designed. If redundant elements are included in a system, a voting arrangement based on the results of an offline or pseudo-online self-test can replace faulty elements by automatically switching in the redundant parts. Looking to the future, when systems on-a-wafer [17] become more of a reality, it will be possible to use a similar redundancy arrangement to permantly configure a wafer with extra processing elements to utilise only devices which pass a self-test.

5 Conclusions

We have developed and demonstrated a methodology and design approach for the inclusion of self-verification in bit serial signal processing chips. Fault coverage can be as high as necessary, is determined without fault simulation, and overheads are extremely low.

The test development is carried out within the unified framework of a silicon compiler, which streamlines and ensures the integrity of the design process. The resultant chips can become part of a fault-tolerant and/or hierarchically self-testing multichip system, within which on-chip and interconnect faults can be detected, pinpointed and dealt with.

References

[1] R.M. Sedmak, "Design for Self-Verification: An Approach for Dealing with Testability Problems in VLSI Designs", Proc. IEEE Test Conference, 112-120 (1979).

[2] R.M. Sedmak, "Implementation Techniques for Self-Verification", Proc IEEE Test Conference, 267-278 (1980).

[3] B. Konemann, J. Mucha and G. Zwiehoff, "Built-In Logic Block Observation Techniques", Proc IEEE Test Conference, 37- 41 (1979).

[4] D. R. Resnick, "Testability and Maintainability with a New 6K Gate Array" VLSI Design, Vol 4 No 2 (1983).

[5] R. A. Frohwerk, "Signature Analysis, a New Digital Field Service Method", Hewlett-Packard Journal, May 1977.

[6] S.W. Golomb., "Shift register Sequences", Pub Holden-Day Inc. San Francisco (1967).

[7] E.J. McCluskey and S. Bozorgui-Nesbat, "Design for Autonomous Test", Proc IEEE Test Conference 15-21 (1980).

[8] J. E. Smith, "Measures of the Effectivness of Fault Signature Analysis", IEEE Trans. Comput. C29, 510-514 (1980).

[9] P. B. Denyer, D. Renshaw and N. Bergmann, "A Silicon compiler for VLSI Signal Processors", Proc 8th European Solid-State circuits conference, 215-218, Brussels (1982).

[10] P.B. Denyer, and D. Renshaw, "Case Studies in VLSI Signal Processing using a Silicon Compiler", Proc. Int. Conf. on Acoustics, Speech and Signal Processing, Boston. 939-942 (1983).

[11] N. Bergmann "A Case Study of the FIRST Silicon compiler", Proc, 3rd Caltech Conference on VLSI, 473-430 (1983).

[12] V. D. Agrawal, "Sampling Techniques for Determing Fault Coverage in LSI Circuits", Journal of Digital Systems 5, 189-202 (1981).

[13] J. Savir, "Random Pattern Testability", 13 Int. Symposium on Fault-Tolerant Computing, Milan (to be published, 1983).

[14] R. F. Lyon, "Two's Complement Pipeline Multipliers", IEEE Trans. Communications 24, 418-425 (1976).

[15] D. Renshaw, Edinburgh University Internal Report (1983).

[16] C. Mead and L. Conway, "Introduction to VLSI Systems", Addison-Wesley (1980).

[17] S.L. Garverick and E.A. Pierce "A Single Wafer 16-Point 16-MHz FFT Processor", Proc IEEE Custom Integrated Circuits Conference, 244-248 (1983).

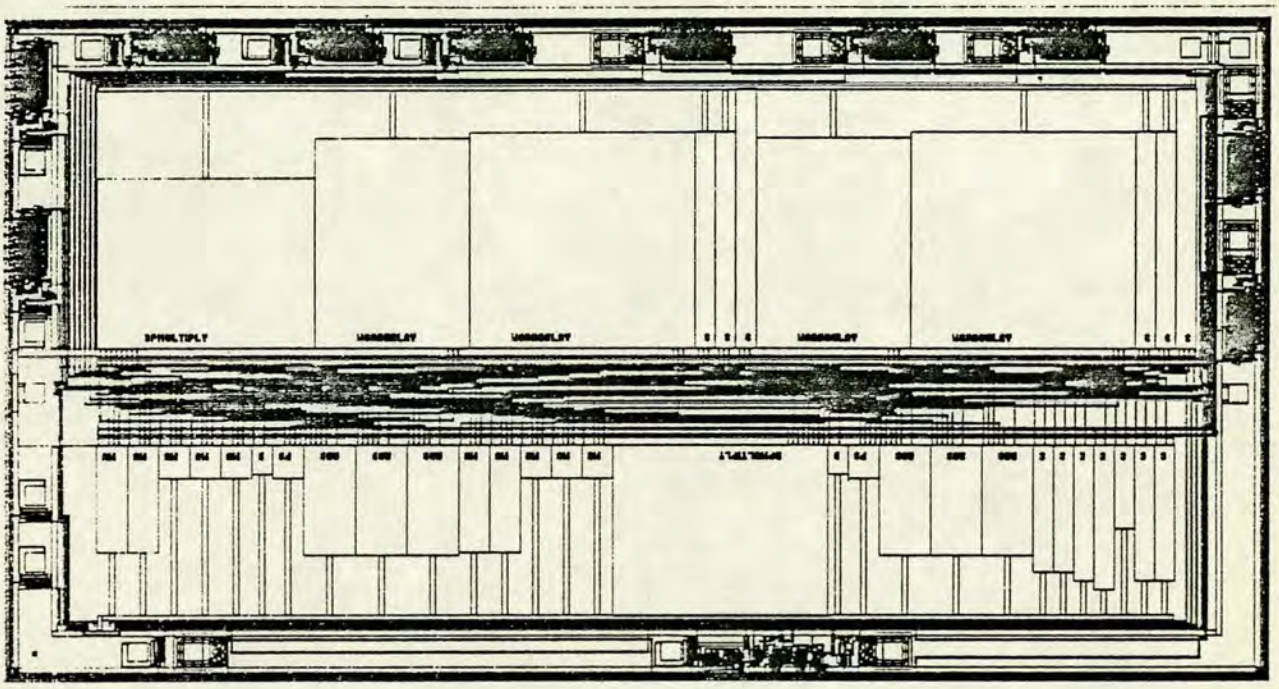


Figure 9: Floorplan of self-testing design (64-point cascadable finite impulse response (FIR) filter).

increased by around 4%. This might be expected to decrease the yield by about 1%.

Each PRBS/FRODO register will have a worst-case power consumption (from analogue SPICE simulation) of 5mW, and about 8mW for the timing circuitry. The overall power overhead for the trial design is thus under 60mW. With an estimated overall chip power consumption of 700 mW this represents a 9% overhead. This could be reduced at the expense of speed performance or by powering down the test circuitry in run mode, but it should be emphasised that this is a worst-case figure.

If the complete system does not have to be self-testing, no increase in system complexity is necessary, as the chip self-test may be initiated manually. In the more likely event that some degree of system self-test is required, a controller chip or module will be required, to define the mode of operation (test or run) and to act upon the results of the self-test. The details of this controller will depend upon the way in which the result of the test is to be used. This topic will be pursued in section 4.

4 System Implications

In section 3 we described a methodology for including self-test, at the chip level, in bit serial processors. There are clearly several stages in the device's lifetime at which the ability to verify its own fault-free status is valuable. We shall discuss only the implications as far as the system designer is concerned.

It is possible to envisage at least three modes of operation within a multichip system. In figure 10 we illustrate the configuration in which such a system will perform its normal processing function. Switching all the chips to 'test' mode allows a chip-test to be performed. Less obviously, switching the first chip in the pipeline to 'test' mode and all the others to 'run' mode configures a system test, within which pseudorandom patterns propagate through the entire chain of processors and their interconnects to yield a 'good system' signature in the final FRODO register. This is a very powerful technique, testing not only the processor's correctness but the integrity of the interconnect, which generally accounts for a high percentage of failures. Flushing out of the system is not necessary if the chip test is performed first to flush out the individual chips.

This two-level test could be expanded to test many levels in a system's hierarchy, with

In designing the test length, therefore, we need only consider the most random pattern resistant operator in the processor. This can be identified by studying a series of graphs such as figure 7 taken from full fault simulations of each operator, which show the clock cycle in which the single stuck faults actually appear at the operators' output. Although this resembles closely the activation curves of figure 6, it actually represents a much more significant result (both computationally and in content). From the graph it can again be seen that the sequence length chosen (1023 bits) ensures 100% coverage of single-stuck-faults in the multiplier and thus a similar degree of coverage for the less random pattern resistant elements. It can also be seen that considerably shorter sequences could be used (say 255 bits) if only 98.5% confidence is required. We have thus reduced the problem of test pattern design to that of assessing, from the testability characteristic graphs for the operators in the processor (c.f. figure 7), the length of PRBS needed to provide any desired degree of coverage. The testability issue can therefore be viewed at a system level, in a manner consistent with the steps involved in arriving at an operator network and flow diagram. The testability of each operator need only be measured once by full simulation to produce its testability graph, to which the system designer may then refer.

3.4 A Trial Implementation

The particular processing function chosen for a trial implementation is not important except that it should be characteristic of the class of bit-serial operators. The 64-point cascadable FIR filter section chosen is typical of the genre of functions for which the bit serial architecture is ideal and at which the FIRST compiler is aimed.

From figure 3 it is clear that the fixed floorplan of a FIRST chip left a significant area of unused silicon in the pad channel. It is a characteristic (indeed one of the most attractive characteristics) of bit serial elements that the communication overhead is low and that the gate/pin ratio is consequently high. It is likely, therefore, that a less fixed floorplan would still result in unused silicon in the periphery of a bit serial signal processing element. We have therefore designed PRBS and FRODO elements and the timing logic in such a way that it can be placed in the pad channel of the FIRST chip. The timing circuit forms an extra entity to be inserted, but the PRBS and FRODO registers can be incorporated with minimal difficulty by simply redefining the signal input and output pads. We have designed all these elements using / design rules [16], with / = 2.5 um, and we present the layout of a FRODO register, complete with signature comparator, in figure 8. This is a conservative design approach and the use of real design rules and a more competitive process would result in much smaller circuit elements. Nevertheless, the entire test circuitry can be placed in the pad channel of our trial design with ease (figure 9). It should be noted that the chip size of the design has not been increased at all by the self-test circuitry.

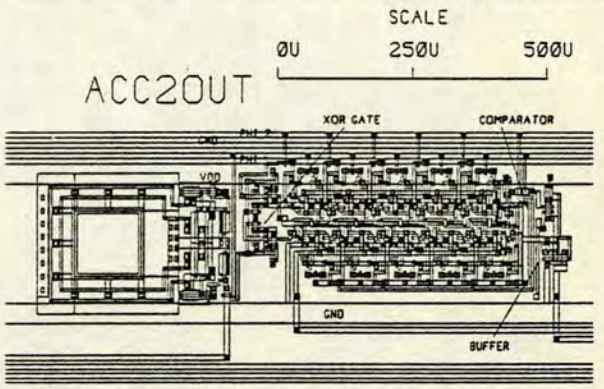


Figure 8: Layout of FRODO register.

We have shown that very high levels of fault coverage, and thence reliability, can be achieved. It remains to us now to count the cost.

3.5 Cost

It is not a trivial matter to persuade designers to conform to restrictive rules or otherwise add to the difficult enough task of designing a chip. Whether a system is designed manually or using a compiler, the design difficulty is not significantly increased by our methodology for the inclusion of self-test, provided a 'standard-cell' approach is adopted. In other words, if a chip is regarded as a collection of standard layout cells connected together, it is a simple matter to redefine the input/output pad cells to include PRBS/FRODO registers respectively, and to include a simply-connected timing cell.

Silicon is precious even as VLSI densities become realities, and a heavy cost in area and therefore yield is not tolerable. We have seen that the chip area is not increased at all by the inclusion of self-test, and that the active silicon area is

signature analysis rather than the more normal 16-bit length. The LFSR polynomials for coefficient and multiplicand generators are 10000001001 and 10010000001 respectively.

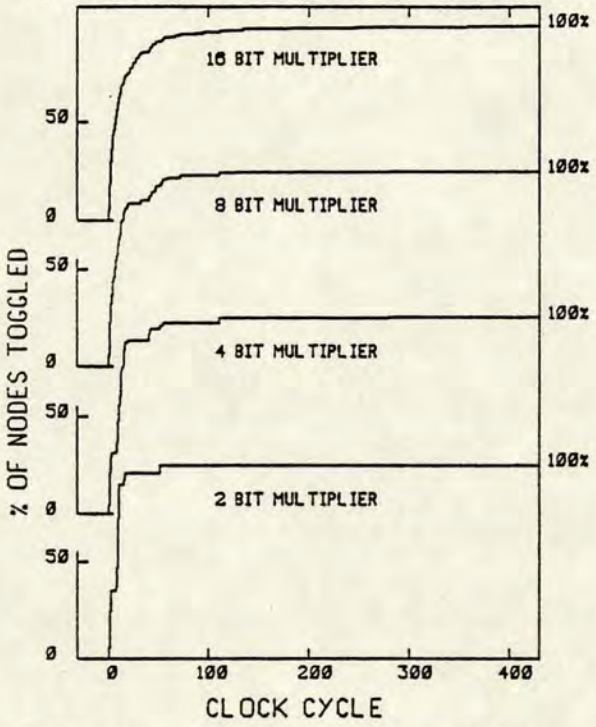


Figure 6: Nodal activity in a modified Booth's algorithm multiplier under pseudorandom stimulation.

To give an insight into the nodal activity within a multiplier under pseudorandom stimulation, we present in figure 6 the results of simulation measurements on 2, 4, 8 and 16-bit multipliers. The vertical axis represents the percentage of all nodes which have toggled (undergone a change of state). The rapid rise between 0 and 20 cycles represents the 'flushing-out' of the shift registers and much of the adder circuitry, while the slower rise to 100% activation by 300 cycles (414 for the 16-bit operator) represents the exercising of the recoder logic, and its resultant effects on the adder. The similarity between these "activation curves" for different multiplier lengths is striking, if not surprising in view of the aforementioned modulatory and propagative randomness property of the architecture. The major effect of increasing the multiplier length is to smooth the curve, as the presence of a larger number of nodes reduces the relative significance of any individual event. The similarity between these curves is very important as it allows us to draw conclusions from the more detailed simulation of a chosen length of multipler which are relevant to all reasonable lengths of multiplier. It also confirms the propagative randomness property, as any significant deviation from randomness at the input to any multiplier module would affect the activation curves of figure 6 by changing the pattern of activity in subsequent modules. This would be manifest as a slowness, or even failure to reach the point where all the nodes have been toggled.

The fact that a given pseudorandom sequence causes a transition (or transitions) on a given node is a necessary but not sufficient condition for both types of stuck-at fault to be detected by the sequence. This is because there are, in general, "don't-care" states with respect to any given node, and the effect of a fault may not be propagated to an output to appear as an error. We have therefore performed an exhaustive fault simulation for the 8-bit multiplier (a useful length, and one which yields tolerable simulation times). The coverage of single stuck-at faults is 100%. With a 10-bit signature analyser the net rate of fault capture will be 99.9%.

We are committed, by the design methodology for self-test, to ensure that the property of propagative randomness is obeyed by all operators. If this cannot be arranged, a minor reconfiguration of the operator may be necessary in test mode to restore the randomness of its output (e.g. the operators output may be XOR'ed with its random input to restore randomness). With the further knowledge that fault free operators obey the well-known 'rubbish in - rubbish out' maxim, we can be confident that a single fault in (say) a multiplier will cause an error at the multiplier's output which will be propagated through subsequent fault-free operators to show up as faulty data at the signature analyser.

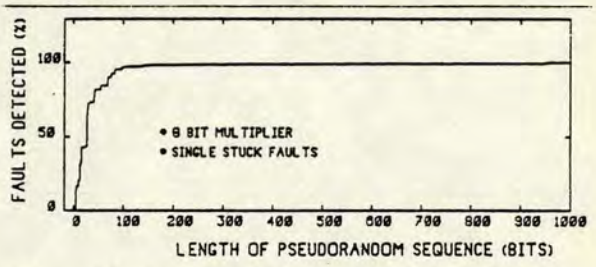


Figure 7: Error detection pattern in an 8-bit modified Booths algorithm multiplier.

is, however, the fault coverage achieved.

The degree of coverage is generally measured by performing a large number of switch-level circuit simulations. First, a 'good machine' simulation must be performed with a particular set of input data to produce a set of good output data for reference. Subsequent simulations will be performed with stuck-at-one and stuck-at-zero faults injected at each circuit node in turn, and the same input data. A fault is said to be 'covered' by the given input stream if its injection causes the output data stream to differ from that of a good machine.

It is obvious that this procedure is computationally intensive and therefore very expensive and time consuming for all but the simplest circuits. Methods are being developed which will reduce this overhead by only injecting faults on 1000-2000 of the circuit nodes [12]. This smaller number of simulatio s will still produce a highly accurate estimate of the fault coverage if the sample nodes are carefully chosen, but long simulation times are still required. Other te hniques are based on assessing the probability that a fault on each circuit node is stimulated by random pattern inputs and subseque tly propagated to an observable output point [13]. These techniques are very exciting as far as random pattern testability of general networks is concerned. For the particular case of bit serial signal processors however, the property of propaga ive randomness exhibited by the ope a ors allows us to adapt a semi-statistical approach in keeping with the hierarc compiler philosophy. We assert that it is only necessary to ensure that individual operators are well tested by our chosen length of random pattern, and that the propagative randomness property is preserved. This impacts upon the design of the building blocks but does not restrict the system designer at all. The most complicated and useful operator in the FIRST system is the multiplier, and we have used it in our trial design. Therefore we have chosen to study the multiplier in detail. In fact, it is clear that the delay operators (shift registers) in figure 2 will be exhaustively tested. The only other operator used is the adder, which is, as we shall see, included as a subprimitive operator in the multiplier. We can therefore be sure that the fault coverage measured for the multiplier represents a lower bound on the coverage for our 64-point filter.

We have studied a bit-serial multiplier, implemented using a modified Booth's algorithm to reduce computational latency [14,15], as shown schematically in figure 5. The design is modular, in that the two-bit element shown in figure 5 may be cascaded to give 2n-bit coefficient lengths. Each two-bit element contains around 200 transistors, the exact number depending on the particular details of the implementation. The hierarchical nature of the design approach is clear even from this subprimitive circuit element, as it can be seen to contain the same add/subtract module that would be used to perform a straightforward add/subtract function.

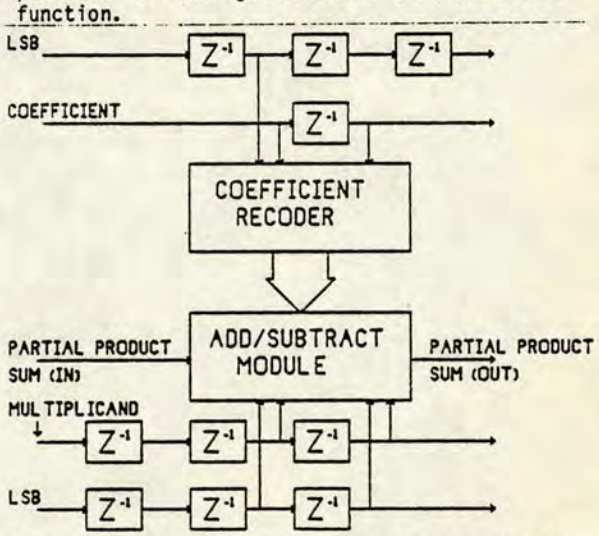


Figure 5: Schematic representation of one stage of a bit serial multiplier using a modified Booth's algorithm.

Multiplication proceeds by recoding 3 contiguous bits of the coefficient word every 2n cycles, (where 2n is the coefficient wordlength). This recoded bit pattern is used to control the calculation of partial product sums in the adder/subtracter block, which are subsequently passed to the next module for further summation. The overall computational latency (delay between input and output) is 3n + 2 bits. The shift registers are exhaustively tested by any nontrivial input sequence longer than 3n + 2 bits. The adder block has some input pattern dependence, however, and the recoder only changes state every 2n clock cycles. One might therefore feel that the multiplier would be resistant to random pattern testing. It is alarming to note that for (say) an eight bit multiplier, there are 2^{16} possible combinations (at least) of coefficient and multiplicand. We shall show that even a small random subset of these combinations (around 70 words) detects a very high percentage of single stuck faults, and thus represents a reliable self-test.

We have chosen to use 10-bit LFSR's for both pseudorandom sequence generation and

predictable state for the test to begin. We shall adopt the approach that test length, and therefore PRBS length, will be determined from high-level considerations in a manner similar to the construction of the data-flow diagram and timing of figure 2. The details are explained and the approach justified in section 3.3

There are two phases of the test. During the first phase, the processor is 'flushed-out' by PRBS stimulation to set all the internal registers to known states. On completion of this phase, the signature analysers are 'switched-on', and the actual test proceeds until fault coverage is adequate.

The test sequence length is determined by the computational latency of the processor (which impacts upon the 'flushing-out' time) and on the engths of PRBS necessary to adequately exercise the circuitry. The latency of our chosen test vehicle is 32 x 14 - bit words, so the flushing-out time must be > 448 cycles. In fact we will see that a ten-bit LFSR for both stimulation and signature analysis gives an adequate PRBS length (1023 bits for flushing out, and a further 1023 for the test proper), and a high rate of fault capture at the signature analyser (99.9%). It should be emphasised that this is merely a convenient length in this case, and in no way precludes the use of longer (or shorter) LFSRs in a future application. For a system clocking at the current maximum rate of 8.3 MHz for a FIRST chip, this represents a test cycle of 0.25ms duration.

In figure 4 we show in some detail how the chosen methodology has been implemented. This diagram is essentially an expanded version of figure 1. It can be seen that a definite sequence of transitions (unknown - 0 - 5V) should result at the 'Test Out' output. This alleviates partially the problems of faulty test circuitry causing bad chips to appear to pass their self-test, although good devices with (say) a faulty flip-flop in the test circuitry may be be rejected. As the self-test circuitry occupies only 4% of the chip area, the probability that a chip is good and the self-test circuitry is faulty is around 1% (for 30% overall yield). It has been assumed that the 'Test In' 0-5V edge is asynchronous, to preserve generality.

A failed self test will occur if one or more of the signatures at the output nodes does not match the expected signature. Failure of the test circuitry to produce a 'Test Complete' pulse will also result in a failed self-test. Grounding the 'Test In' pin puts the chip in 'run' mode. It would be possible to arrange for the test circuitry to be powered down in run mode, although its low power consumption scarcely warrants this extra complexity.

3.3 Coverage

We have shown how self-verification can be incorporated into bit-serial processors with minimal extra design effort. The 'bottom-line' for any testability enhancement

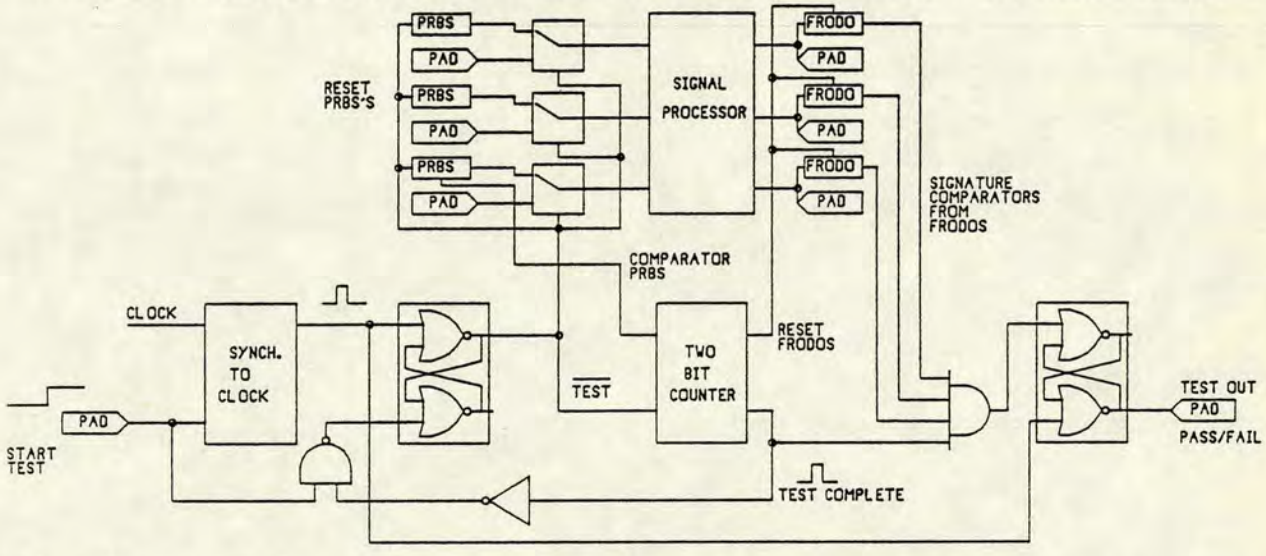


Figure 4: Operation of the self-test circuitry

can be derived from signal processing algorithms by the use of a set of synthesis rules. These will ensure, amongst other things, that all loops are broken by a multiplex point (to reduce sequential depth to testable proportions).

The resulting coded file can then be compiled for layout and simulation. Floor plans give chip sizes to aid partioning and the simulator allows verification of the system source description. Each of the compilers incorporates extensive checking and diagnostic warning for illegal constructs, as an aid to design. These include, e.g., checks on syntax, parameter values, fan in, fan out, undriven or unconnected nodes, synchronisation etc.

Efficient automation of mask geometry is achieved by the use of constrained architectures, which generate the floor plan. The actual format for the floor plan is determined also by the constraints of the technology. An example of a FIRST floor plan for a five micron polysilicon gate NMOS process is shown in figure 3. The main features are a single central wiring channel off w ich bit serial primitives are placed, and into w ich they communicate. The most important feature here is the low communication area which bit serial systems permit and the consequent ease of partioning which this allows.

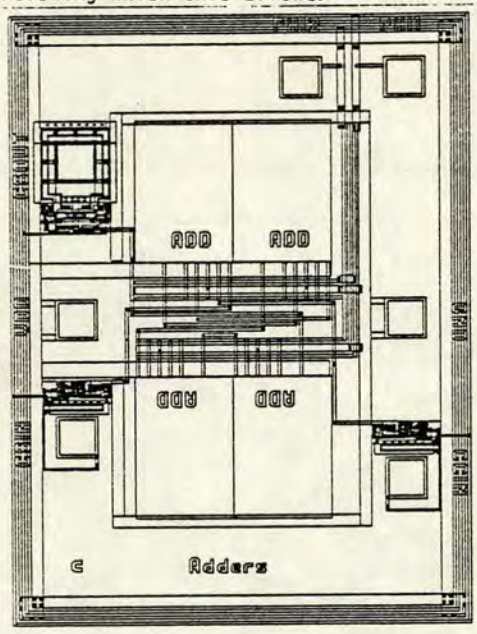


Figure 3: Floorplan of a simple FIRST chip.

The layout is constructed from a proven set of circuit blocks, which are assembled into the primitive operators of FIRST according to the parameters given in the high level specification. For instance, blocks each representing two binary bits of multiplication are combined to give multiplication of binary words of length 2n (n = 2, 3, etc). From the viewpoint of testability, the principal features of these primitive operators are that even the most complex exhibits low average gate fan-in, and that most can be shown to propagate random signals. In other words, random input sequences lead to random output sequences. If an operator violates this property of propagative randomness it can be modified by means of a multiplex point to restore randomness. As a consequence of these features, it is possible to have confidence in the reliability of random pattern testing in systems of interconnected primitives without fault simulation of the entire system, provided the component primitives have been shown to be so testable.

3.2 The Design Methodology

As discussed in section 1, we have set out to define a means of approaching the design tasks embodied in the FIRST silicon compiler which will result in a verified processor where a self-test capability is included automatically. The compiler ensures correct functionality and layout integrity, and the self test methodology must allow design to be performed to give any reasonable desired degree of confidence in the self-test. The hallmark of a successful methodology is that it can be easily understood and applied by system designers who are not expert in test/self-test methods. We have developed such a methodology.

The primary virtue of our approach is that the architecture and functionality of the central system are substantially unaltered. The only constraints are that looped data paths must be openable, and that access can be gained to such opened loops for inspection of the pseudo-output point generated by the break. In this way long time constants associated with feedback loops are avoided. This increases the system's latency, (pipelined delay between input and output) and determines its value. Feedback loops can often be broken by existing control inputs, and the inclusion of circuitry to break those which cannot should not represent a serious overhead or added complexity.

Control signals are either operated in the normal mode or used to force open loops as described above. Although this results in an incomplete test of some of the control circuitry, it is necessary to ensure a

Register Output Data Observer (FRODO) to distinguish it from a PRBS register. With this nomenclature, we can represent the proposed self-testing system by figure 1. Switching the multiplexer from PAD to FRODO configures the device in 'test' mode.

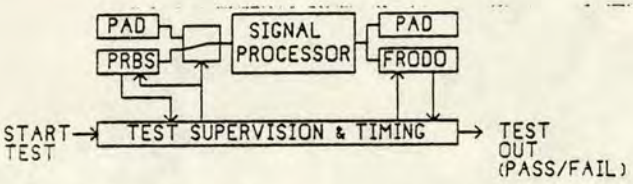


Figure 1: Schematic representation of a PRBS/signature analysis self-testing system.

The self-verification capability has been implemented at chip level, such that individual devices are capable of self-test. It will be seen that this enables a hierarc ically structured system self-test covering on-chip and interconnect failures.

Cost and coverage are areas of concern regarding self-test. We will show that the cost in our approach is very low. Coverage is reduced by two factors. The first, and most significant, is the incomplete exercising of the circuit's logic, and the second is he less than perfect rate of error capture by signature analysis [5,8]. We will not address directly the latter problem, except to note that the use of maximum length FRODO registers maximises the rate of capture of both independent and 'repeated-use' errors [8]. The nontrivial problems of 'burst' and other classes of error we leave to the experts. We do address in section 3.3 the question of fault coverage by random patterns, developing a set of design rules for ensuring a very high degree of coverage of the (as yet irreplaceable) single stuck faults.

3 Self Test for Bit Serial Signal Processors

3.1 Bit Serial Signal Processing and FIRST

Custom VLSI offers many advantages for the implementation of real-time signal processing systems. These advantages include small size, low power dissipation, and good product security. The principal disadvantages have been high costs and long design cycles. As VLSI technology has been developed, the principal problem has become that of design and the management of design complexity. The solution of this problem lies in the evolution of appropriate CAD tools, which offer a way of handling complexity, and reducing design cycles and hence cost.

Two main approaches have developed. One is based on graphics design tools backed up by post design analysis of mask geometries for initial debug, design checking and correction. More recently, an alternative approach has been developed, namely that of silicon compilation, which allows a high level program specification to be compiled correctly into the appropriate corresponding mask geometries. The normal minimum requirements of a silicon compiler are that it offers a high level input language, a mask geometry compiler and a guarantee of correctness. Most silicon compilers also seek to offer a simulator which models the compiled design behaviour. A further stringent requirement is that mask geometries must use "silicon real estate" reasonably efficiently. A further condition, not usually considered but in our view essential is that all structures generated must be testable and must have an economic yield.

A silicon compiler called FIRST is in development in Edinburgh. First offers -

a) A methodology for mapping signal processing algorithms into bit serial networks, including rules which guarantee ease of testability.
b) A high level language and compiler for specifying hierarchical net lists of primitives.
c) A mask geometry compiler.
d) A behavioural simulator.
e) Automated generation of test patterns during design.

Since details of FIRST have been reported elsewhere [9, 10, 11], only those features relevant to the present paper will be summarised.

The algorithm, or signal processing function to be implemented can be represented as a signal flow graph, an example of which is shown in figure 2. This flow graph can then be coded concisely using the FIRST high level language. Correctly formulated flow graphs

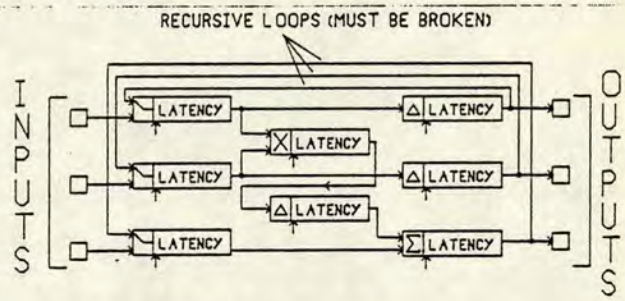


Figure 2: Section of a signal flow graph for a bit-serial signal processor.

Self - Testing in Bit Serial VLSI Parts: High Coverage at Low Cost.

Alan F. Murray*, Peter B. Denyer† and David Renshaw†.

* Wolfson Microelectronics Institute † Dept. of Electrical Engineering,
University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL.

ABSTRACT

This paper presents a methodology for the inclusion of random pattern/signature analysis self-test in bit serial signal processing chips, within the unifying framework of a silicon compiler. Fault coverage is very high, and is determined without full fault simulation. The cost in silicon, power, complexity and design difficulty is extremely low, as is shown by a trial design. A hierarchical system test can be performed, leading to the possibility of fault tolerance.

INTRODUCTION

Self-test is widely accepted as a valuable tool for preventing the unrestrained growth of test time and cost and for enabling the design of fault-tolerant systems[1]. Much work has concentrated on generalised techniques for the inclusion of self-test [2,3] and there are successful examples of particular self-testing systems (mostly microprocessor but more recently, gate arrays [e.g. 4]). In this paper we describe a rather different approach, in which the adoption of a particular architecture for a broad class of systems (signal processors) allows the development of a coherent self-test strategy. We show that this approach leads to a design discipline within which fault coverage can be estimated accurately without fault simulation, and the self-test circuitry placed automatically by a high-level silicon compiler. The self-test method is the well known combination of pseudorandom stimulation and signature analysis.

In section 2 we describe briefly the advantages of self-test as a route to testability and of signature analysis as a self-test methodology. In section 3 we give details of the chosen signal processor architecture and an associated silicon compiler. This is followed by a discussion of the inclusion of the self-test capability and the fault coverage achieved, along with a trial chip implementation to final artwork stage of a cascadable 64-point FIR filter, leading to an assessment of the cost in increased design difficulty, complexity and silicon real estate.

The final section (4) outlines the system implications and conclusions to be drawn from this work.

2. Self-Test, Random Patterns and Signature Analysis

Self-test represents the ability of a system, subsystem or chip to provide a indication of fault-free status with power and clock signals being the only external stimuli required [1]. The advantages of self-test are many and well documented [1,2] and we tabulate them below merely for the sake of completeness.

1) Every test during a device's lifetime is made easier by self-test.
2) Expensive automatic test equipment is made less essential.
3) The addition of other techniques, such as LSSD, is not precluded by self-test and indeed may be necessary under some circumstances.
4) Test proceeds at the system clock rate.

For non-microprocessor products, pseudorandom pattern stimulation combined with signature analysis [5] is rapidly becoming the pre-eminent approach to self-verification. Pseudorandom patterns are provided by linear feedback shift registers (LFSRs) [6,7] and signature analysis uses similar LFSRs to compress the processor's response data to a relatively short signature word or words which indicate whether or not the system is functioning correctly. Pseudorandom binary sequence (PRBS) generators and signature analysers are often implemented as distributed LFSR's using existing latches from within the circuit, which are generally of the reconfigurable BILBO (Built In Logic Block Observer [3]) type. This technique normally results in the most efficient implementation of self-test. For the particular case of bit-serial processors we shall show that the versatility of the BILBO approach is neither necessary nor desirable. Accordingly, we have implemented the PRBS register and the signature analyser as compacted circuit elements seperate from the internal latches. We shall refer to this non-distributed signature analysis register as a Feedback

The IEEE International Test Conference - Cherry Hill, Philadelphia, Oct. 1983.

pp 260 - 268

of the rapid availability of physical chip size estimates. In this case 40% of the system (2 chips) is used to generate the adaptive step-size estimate. This might be considered excessive, and could initiate a reexamination of the algorithm to find a more efficient estimator. At least until the devices are submitted for maskmaking and fabrication, no significant commitment is made to any device; the physical implications of new algorithms can be explored for little real cost.

5. Case Study 2: An Adaptive LMS FIR Filter

Adaptive FIR filters find application as echo, distortion and interference cancellers. For this study we have attemted a speech-bandwidth 256-point echo canceller, using a full (unclipped) version of Widrow's LMS algorithm [5]. In this case the higher computational requirement merits a linear array approach, in which the total filter is formed from a cascade of shorter sections. Accounting for the serial computation rate, we find that each physical bit-serial LMS processing unit may support 32 virtual filter points at a sample rate of 12kHz.

The integrated system partitions conveniently such that one 32-point LMS section occupies a single chip, using 5-micron nMOS technology. Thus the completed filter comprises a cascade of eight 32-point sections (in 16-pin packages), plus one 24-pin device for output summation, and one other 24-pin chip for error computation and system control. The total transistor count is approximately 35,000.

6. Further Case Studies

The integration of several other systems is under investigation with the Silicon Compiler. These include:

- a pipelined FFT
- a large time-bandwidth complex matched filter system
- an implementation of Lyon's computational model of the cochlea [6].
- bandpass wave-digital filter structures.

7. Conclusions

We have demonstrated the potential of the Silicon Compiler as a powerful design tool for custom VLSI systems. Development times are comparable with any high-level software approach, yet efficient custom hardware results. The compiler permits an effective identity between abstact algorithm and physical representation which encourages the exploration of new algorithms with no significant commitment to any device set except the one finally commissioned.

We argue that this approach is a most expedient route to the implementation of complex real-time VLSI signal processing systems, even for low-volume or prototype requirements. Through this work we hope to encourage system designers everywhere to consider silicon as a flexible and attractive implementation medium.

8. References

1. P. B. Denyer, D. Renshaw and N. Bergmann. "A Silicon Compiler for VLSI Signal Processors." *ESSCIRC '82*, Vrije Universitiet, Brussels, September 1982, pp. 215-218.

2. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing." *VLSI 81*, University of Edinburgh, August 1981, pp. 131-140.

3. E. H. Satorius and S. T. Alexander. "Channel Equalisation Using Adaptive Lattice Algorithms." *IEEE Trans. Comm.*, Vol. COM-27, 1979, pp. 899-905.

4. M. J. Rutter, P. M. Grant, D. Renshaw and P. B. Denyer, "Design and Realisation of Adaptive Lattice Filters." To be published in these proceedings, ICASSP '83.

5. B. Widrow, *In Aspects of Network and System Theory*, Holt, Rinehart and Winston, 1970, ch. Adaptive Filters.

6. R. F. Lyon, "A Computational Model of Filtering, Detection and Compression in the Cochlea." *ICASSP '82*, IEEE, 1982, pp. 1282-1285.

Synthesis of this system has proceeded exactly along the lines indicated above. The first step involves mapping the computational processes as bit-serial flow-graphs. For example, Figure 4-1 gives a flow-graph for the simple lattice operation.

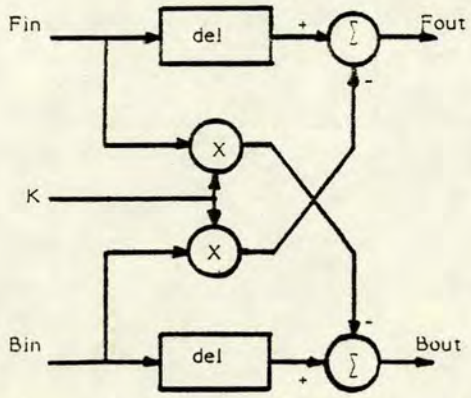


Figure 4-1: Flow-graph of lattice section

The blocks marked 'del' are included to equalise the natural latency through the serial multipliers. The total latency of this stage is estimated as 22 bits, and this becomes the system wordlength, although within this range the expected significance of the F and B signals occupies only the lower 12 bits. For bit rates of 8MHz, the stage computation time is thus 2.75 microseconds. Now we are able to multiplex this single, physical processing unit to effect 16 virtual lattice stages at an external sample rate of 20kHz. To implement this filter, the recursive states identified above are saved and circulated through the lattice computation unit via FIFO delay primitives.

The total system flow-graph is completed by implementing each of the required processes in this way, adding the appropriate multiplexing and control networks. The resulting flow-graphs are then entered to the compiler for simulation and chip generation. The compiler checks for syntax and network integrity before instantiating the finished layouts.

In this case the system is partitioned around its three main functional groups: one chip implements the lattice, one chip implements the parcor coefficient estimator, a chip pair is required to compute the step size estimate, and a final chip contains the central control generator and some remaining multiplexing circuitry. None of the chips exceeds a packaging requirement of 16 pins (this is another nice feature of bit-serial architectures), and the largest device measures 5.1*4.9 mm in 5 micron nMOS technology. Figure 4-2 shows floor plans generated by FIRST for the entire chip set.

The completed system, containing around 16,000 transistors, was commissioned in just four weeks by our 'silicon-naive' designer. The five chip LSI set replaces an original system containing 160 TTL parts, with a power consumption of 10 Watts, which took over six man-months to design and debug. More than any other argument, we hold this as a powerful vindication of the Silicon Compiler approach. Further details of this system are given in [4].

This study also serves to demonstrate the value

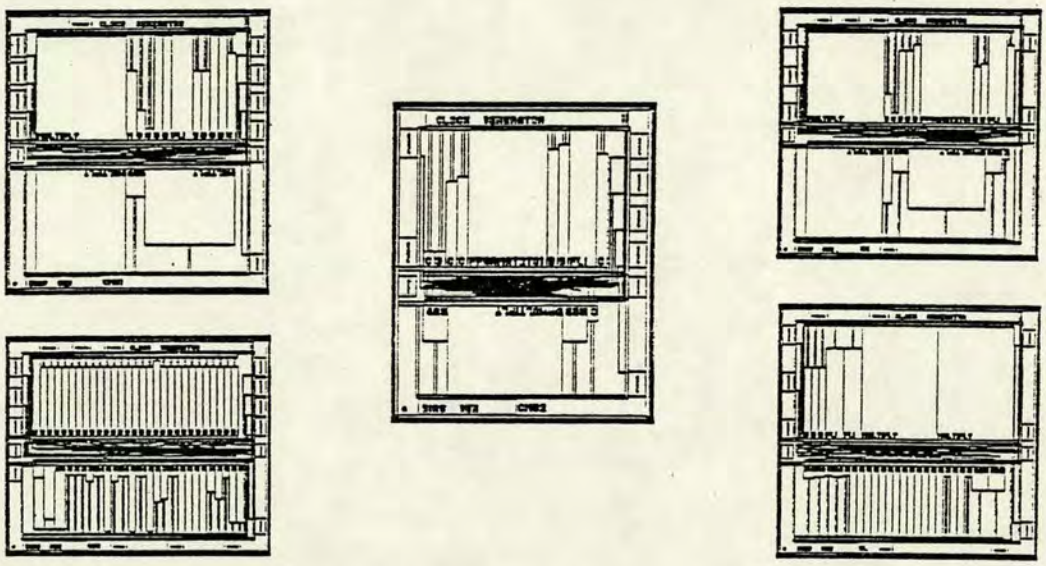


Figure 4-2: FIRST-generated floor plans for the 5-chip adaptive lattice filter.

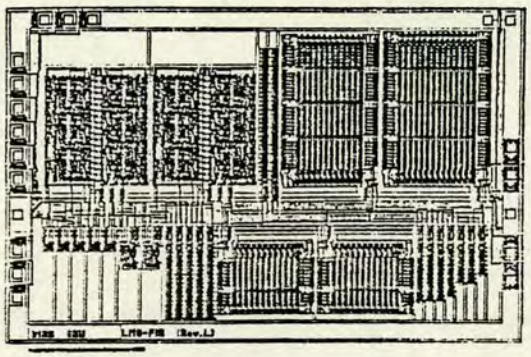


Figure 2-1: A fully instantiated chip layout generated by FIRST.

As part of the first-time-right philosophy, FIRST also supports a system simulator that enables the user to verify and debug his system description, prior to compilation. This simulator runs from the same high-level system description file that is used by the Compiler. This completes the design loop and ensures an error-free system implementation.

3. Designing Bit-Serial Systems

Historically, bit-serial architectures have not received the same attention as bit-parallel schemes. However, through the many advantages that bit-serial systems appear to offer to VLSI implementation, we anticipate that this situation will be rectified. In the meantime, we have found it necessary to develop generalised design methods for real-time bit-serial systems. These involve not only the synthesis of bit-serial computation units, but also higher issues concerning real-time concurrent processing arrays, and at lower levels the effects and containment of word growth in fixed-point systems. To cope with these issues we have evolved an overall methodology whose explanation does not lie within the scope of this paper; nevertheless a summary is appropriate. We identify five steps to system synthesis:

1. Formulation of the signal processing algorithm as a mathematical recursion.

2. Derivation of a computational flow-graph to implement each statement of the recursion.

3. Bit-serial interpretation of the flow-graph, including fixed-point arithmetic effects. Calculation of computational latency.

4. Determination of concurrency required to match computational and signal bandwidths. Implementation of multiplex network and instantiation of physical processing array.

5. Addition of intialisation and test capabilities.

6. System partitioning for optimum chip count and size.

The resulting system flow-graph is then ready for direct implementation (or simulation) through the Silicon Compiler.

We highlight two case studies out of several currently in progress. By coincidence, these are both adaptive systems, although they are selected for no other reason than that they represent more complex system forms than their non-adaptive counterparts. As indicated later, the capabilities of the compiler are extremely general, and are by no means restricted to these systems.

4. Case Study 1: An Adaptive Lattice Filter

Our first case study is also the most definitive, since it was attempted by a system designer with no previous integrated design experience. The adaptive lattice filter which is the subject of this investigation is primarily intended for application as a fast equalizer for modem polling systems, however it may also be used for linear predictive coding (of speech), or as part of a spectral analysis system. The filter is based on a gradient algorithm presented by Satorius [3], with the exception that a division in the power estimator is approximated to first order by a Taylor series expansion. This trick enables a potentially expensive division to be replaced by a multiply and add. The resulting recursive definition of the filter, in terms of stage (N) and time (T), falls into three parts:

Lattice:

$$Fin(N,T) = Fout(N-1,T)$$
$$Bin(N,T) = Bout(N-1,T-1)$$
$$[Fin(1,T) = Bin(1,T) = Sigin(T)]$$

where for all N,T:

$$Fout = Fin - K*Bin$$
$$Bout = Bin - K*Fin$$

Parcor coefficient:

$$K(N,T+1) = K(N,T) + Mu(N,T)*CORR(N,T)$$

where for all N,T:

$$CORR = Fout*Bin+Fin*Bout$$

Step size:

$$Mu(N,T+1) = Mu(N,T)*(1+A-POWER(N,T))$$

where for all N,T:

$$POWER = Bin^2+Fin^2$$

and A is a small +ve constant.

CASE STUDIES IN VLSI SIGNAL PROCESSING USING A SILICON COMPILER

Peter B. Denyer

David Renshaw

University of Edinburgh

Abstract

We advocate a custom approach to VLSI signal processing using a Silicon Compiler. Through the Compiler, system designers with no previous VLSI experience may enjoy all the advantages of an integrated system implementation. At the same time, cost and timescales are dramatically lower than those conventionally associated with custom design. To illustrate these advantages we present case studies of some VLSI systems currently in development at Edinburgh.

1. Introduction

The complexity of many interesting signal processing algorithms prohibits their realisation in real time. Often the required process exceeds the capabilities of off-the-shelf programmable parts, while assemblies of standard MSI/LSI parts can be both physically and economically unattractive. Under these circumstances, we believe that the most appropriate medium for development is custom-designed silicon. Traditionally however this route is characterised by excessively long (and often repeated) design cycles. The enormous commitment that this entails has led to an impression that silicon is an inflexible medium, to be approached only after careful prototyping, and with a large market in view. This impression is contradicted with the advent of the Silicon Compiler, a tool which may automatically, and virtually instantaneously, generate an entire chip design from a concise high-level system description. Such a facility changes our perception of silicon as a development medium. The designer is freed to work at the architectural level, where his creative talents are arguably most effective. The Compiler is then accessed frequently for rapid feedback on physical chip sizes and power consumption.

With no prior knowledge of integrated circuit design, VLSI design cycles around four man-weeks are typical. By virtue of their automatic construction, the systems so produced are error-free, guaranteeing a first-time-right implementation.

In this paper we review the development of VLSI systems using the Silicon Compiler 'FIRST' [1]. We hope to show that custom VLSI systems enjoy not only considerable size and performance advantages over alternative forms of implementation, but also that VLSI design with a Silicon Compiler can be economically attractive, even for prototype systems.

2. The Silicon Compiler

We have developed a Silicon Compiler to support dedicated (as opposed to programmable), bit-serial architectures, following the work of Lyon [2], and others. This restricted architectural choice enables the Compiler to work efficiently, and at the same time encourages the development of a powerful, unified synthesis technique. This is reflected in an effective, structured design style which contributes greatly to a speedy design cycle.

FIRST (for 'Fast Implementation of Real-time Signal Transforms') supports a library of predesigned primitive operators (multiply, add, delay, etc.), which are called by the user as a network, or flow-graph, of the system to be implemented. For example, the user may call and connect four multipliers, an adder and a subtracter to implement a complex multiplier. Such a routine may become a new system operator, to be called in future as required. In this way FIRST naturally supports a hierarchy of:

primitives
operators
chips
subsystems
systems.

Throughout this hierarchy, modules obey rigid communication conventions which make their interconnection both straightforward and failsafe.

FIRST generates chips by instantiating all of the requested primitives and placing them within a target floorplan. The flow-graph network is then implemented by connecting the primitives along a central communication core. It is an advantage of bit-serial systems that, because signals occupy single wires, this core does not dominate the chip area as might be the case with a bit-parallel architecture. After adding pads, clocks, and power supplies the chip is complete. An example of a compiled chip from one of the following case studies (the LMS filter) is given in Figure 2-1. The initial primitive library is designed for 5 micron NMOS processes running at bit-rates of 8MHz. At this technology level each chip typically contains 2,000 to 5,000 transistors and is specified by some 20 to 50 lines of system description. In all cases presented here the upper limit on chip complexity is purely technology-limited.

Proc. IEEE ICASSP'83, pp.939-942, Boston, April 1983.

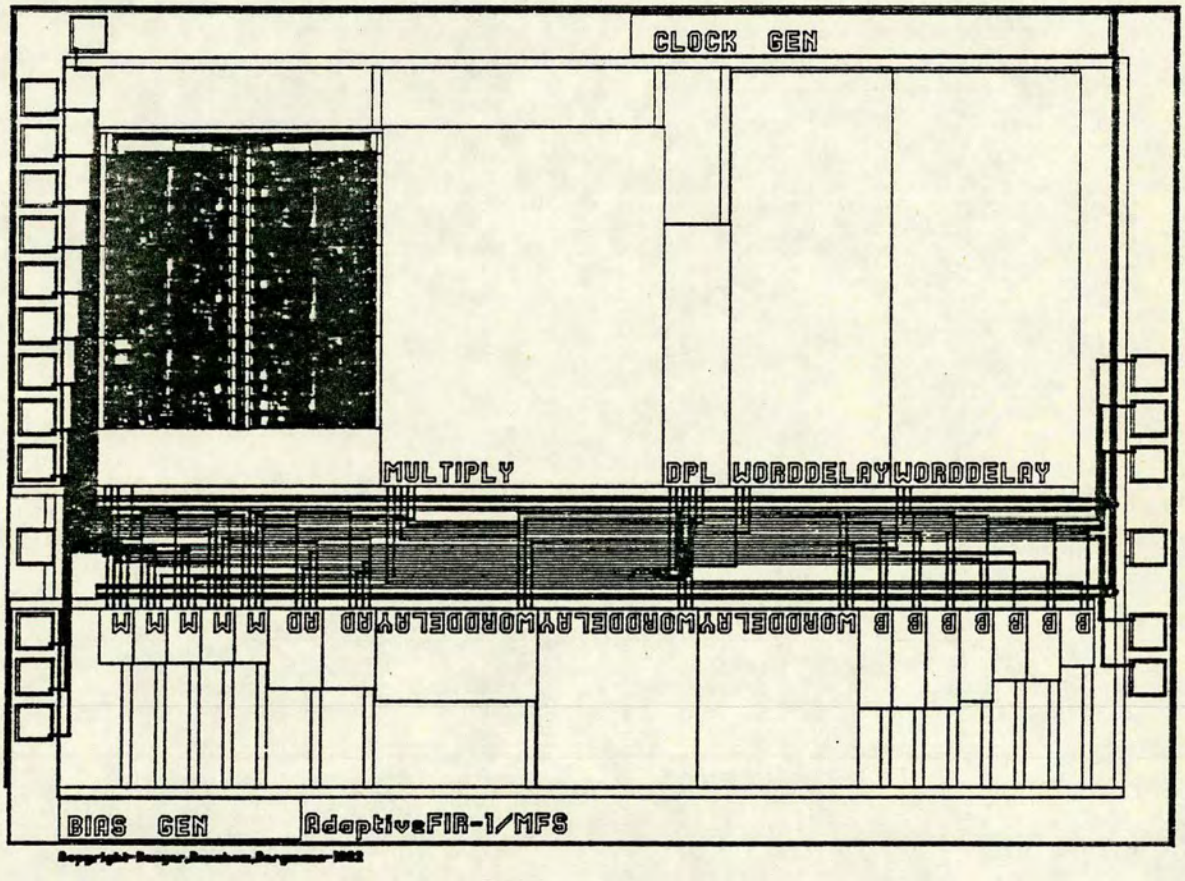


Figure 1. (b) Compiled chip layout (with Multiplier detail).

simulation and design iteration at the system level – but rather as an indication of a virtually instant identity between an abstract system description and its ultimate physical representation.

Conclusion

Silicon compilation offers massive cost and time reductions over traditional LSI design techniques. It also offers the system designer, who may have no LSI design experience, a capability for implementing complex integrated systems with every expectation of first-time success.

We have implemented such a compiler for signal processors using bit-serial architectures. A range of initial examples covering speech, sonar and telecommunication applications have demonstrated the feasibility of this technique for producing efficient chip layouts within development timescales of a few mandays.

Acknowledgements

The authors would like to acknowledge helpful contributions by J. Gray, I. Buchanan and D. Myers.

References

1. Lyon R.F., "A Bit-Serial VLSI Architectural Methodology for Signal Processing", *in VLSI 81*, ed. J. Gray, Academic Press, 1981
2. Mead C., and Conway L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

```
!Adaptive(LMS)Transversal(FIR)Filter
!filter stage of multiplexed filter sections

CONSTANT round=1,limit=1,nolimit=0

CONTROL INPUT c0,c1,init
CONTROL INPUT C0T0,C0T1,C0T2,C0T3,C0T4,C0T5,C0T6,C1T0
INPUT Xin,YMSBin,WMSBin,WLSBin

OUTPUT Xout,YMSBout,YISBout,YLSBout,OVFL

OPERATOR FIR[wordlength,idles,stages]

   SIGNAL s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,
          s16,s17,s18,s19,s20,s21,s22

      Multiplex[1] (c1t0) s3,Xin -> s1
      Multiplex[1] (INIT) s17,WMSBin -> s7
      Multiplex[1] (INIT) s18,WLSBin -> s8
      Multiplex[1] (c0t2) s6,s5 -> s6
      Multiplex[1] (c0t6) YMSBout,YISBout -> YMSBout

      Multiply[round,wordlength-2] (c0t0) s2,YMSin -> s5,nc
      Multiply[round,wordlength-2] (c0t5) s22,s17 -> YLSBout,YISBout
      Add[nolimit,1] (c0t3) s6,s11 -> s15
      Add[nolimit,1] (c0t3) s5,s12 -> s16
      DPLimit[wordlength] (c0t4) s16,s15 -> s18,s17,OVFL

      Worddelay[idles+1,wordlength-2] (c0t0) s1 -> Xout
      Worddelay[idles+stages-1,wordlength-2] (c0t1) s7 -> s19
      Worddelay[idles+stages-1,wordlength] (c0t1) s8 -> s10

      Constant half = stages/2,
               rest = (stages - 1) - half
      Worddelay[half,wordlength-2] (c0t0) Xout -> s4
      Worddelay[rest,wordlength-2] (c0t0) s4 -> s2

      Bitdelay[wordlength] s1 -> s19
      BItdelay[wordlength] s19 -> s20
      Bitdelay[wordlength] s20 -> s21

      Bitdelay[wordlength-1] s2 -> s3
      Bitdelay[wordlength-4] s9 -> s11
      Bitdelay[wordlength-4] s10 -> s12
      Bitdelay[wordlength/2+1] s21 -> s22


   END

! ALLOCATE VALUES TO PARAMETERS : wordlength,idles,stages


FIR[14,10,47]

ENDOFPROGRAM
```

Figure 1. An example of silicon compilation using FIRST. Figure shows one part of a 3-chip LSI set for adaptive speech echo cancellation.
(a) FIRST system description file.

to compose more powerful operators (FFT, Biquad, etc.) by procedural definitions of primitives. Once composed, these can be made available to other users, rather like a library of useful subroutines.

Table I lists the primitive operators currently supported by FIRST; it is desirable that this list be as short as possible. Table II gives a library of the procedural definitions that have been generated to date.

Table I: Primitive Operators

Multiply
Add
Subtract
Limit
Scale
Multiplex
Sort
Absolute
Bit delay
Word delay

Table II: Procedural Definitions

Biquad
FIR
Adaptive FIR
Lattice
Adaptive Lattice
Complex to Magnitude
Complex Multiply
Butterfly
etc., etc.

The primitive operators are parameterised to include functional options – for example in the Multiply primitive these specify the coefficient word length and the output format (rounded or truncated). These parameters determine the composition procedure that will be used to construct the operator during assembly. In general, parameters may be specified as explicit numbers, variables, or arithmetic expresions. Thus it is also possible to parameterise the procedural definitions. For example, the Biquad procedure will compose a set of M filters, each containing N second-order sections, with data words of d bits and coefficient words of c bits.

Internally, FIRST supports two physical descriptions of each primitive operator; geometric and behavioural. The geometric description contains CIF code for a collection of hand-packed cell layouts that are assembled by the compiler to form the operators. The behavioural description takes the form of bit-level procedures for perfect functional simulation. (The clock period is a satisfactory minimum time metric, and the boolean logic levels are a suitable amplitude metric, once the primitives have been functionally proven at the desired maximum bit rate.)

There is scope within the FIRST system to accommodate automatic test pattern generation, although we have not yet implemented this feature.

FIRST time right

The simulator and compiler work from the same system definition input file. Under these conditions it is possible to guarantee that the physical implementation will match the input system description, and also that the simulation will match the physical implementation. Together these features offer a level of confidence that existing design techniques have never permitted.

FIRST Implementation

We have implemented, and are verifying, a library of primitives (Table I) for 5 μm nMOS processes, operating to a fixed-point 2's complement format with a maximum bit-rate of 8 MHz.

An Example

Figure 1 shows one device from a three-part LSI chip set that implements an adaptive FIR filter for automatic echo cancellation in real-time speech systems. The merit of the concise system description language is evident in the short FIRST input file shown in the upper half of the figure. This is typical for an LSI part, and clearly indicates the potential of this approach for handling VLSI systems of much greater compelxity.

A plot of the resulting chip, generated by the FIRST compiler from the system description file, is shown in the lower half of the figure. The operators are indicated on this plot by bounding boxes, but to give some idea of the efficiency of the layout one of the operators (a Multiplier) has been plotted in full. It is evident from the plot that very little silicon area is wasted (we estimate 20-25% typically) – an interesting result for critics of automatic layout who point to irresponsible use of silicon area.

Experience to date in the design of four systems, totalling 10 LSI parts, suggests a timescale of the order 2 mandays per part for this type of system development, given a base level understanding of signal processing with fixed-piont arithmetic. No knowledge of MOS circuit design is required.

We do not suggest this timescale as a future guideline – a much greater allowance must be made for

A SILICON COMPILER FOR VLSI SIGNAL PROCESSORS

P.B. Denyer, D. Renshaw, N. Bergmann
Departments of Electrical Engineering and Computer Science, University of Edinburgh, UK.

Abstract

This paper reports a silicon compiler that allows the rapid implementation of LSI/VLSI signal processors from a high level system description language.

Introduction

The advantages of custom LSI design have, in the past, been offset by the high costs associated with long development timescales and multiple design iterations. These overheads make custom design unattractive for medium and low volume markets, and for products requiring rapid development. Moreover, the ad hoc design techniques in use today cannot be extended to cope with VLSI complexities without greatly exaggerating these disadvantages.

Past attempts at overcoming these problems have lead to fragmented solutions which only in small measure diminish the overheads, and often introduce further problems of their own. A major disadvantage usually lies in the requirement for a high degree of MOS circuit design and layout skill, making it difficult or impossible for the systems designer to implement new structures in silicon. Typically, he may build a breadboard of SSI, MSI and LSI parts to prove the system before investigating its integration. This is a restrictive practice that prohibits the proper exploitation of silicon as a development medium for a host of new VLSI systems and architectures.

An alternative approach is to automate the entire implementation procedure, and offer a single system-level interface to the user. Such an automatic design system, coupled with rapid, low-cost maskmaking and fabrication, should encourage the development of prototype systems in integrated form, dramatically cutting the total development cost and timescale. In addition to these economies of cost and time, it may also be possible to guarantee a correct implementation by virtue of automatic construction.

This paper reports on such a 'silicon compiler' for LSI and VLSI signal processors. It is called FIRST – Fast Implementation of Real-time Signal Transforms.

A Framework

To obtain results in the form of efficient chip layouts, the compiler must be tailored to a complete design style, incorporating specific architectural, topological, timing, circuit and layout conventions. The FIRST compiler is based on such a methodology for bit-serial architectures, reported recently by Lyon [1].

Bit-serial processors generally offer optimal area-time tradeoff and are compact enough to allow concurrent implementations of complex systems within reasonable chip areas. The overhead costs of communication and control are minimal, allowing the majority of the chip area to be devoted to the computational processes. System partitioning into multiple chip sets is eased by the single-wire serial communication format. In addition, a variety of multiplexing levels may be used to properly match signal and processor bandwidths for real-time applications.

FIRST uses a fixed floor-plan format based on a central communication channel that supports ranks of bit-serial processors to either side. Each of these processors obeys strict geometric and electrical conventions at the wiring channel interface, but is otherwise unrestricted in form. This well-defined interface has enabled us to delegate the implementation of individual processors to a range of designers. It also allows new processors to be added easily at any time.

The processors operate synchronously from a global 2-phase non-overlapping clock that is distributed from the edges of the communication core.

As in [1] the nMOS circuit, layout and timing conventions suggested by Mead and Conway [2] have been adopted for the lower levels of the design hierarchy, though it is possible to replace these to suit other technology variants.

System Design and the Silicon Compiler

FIRST supports a high-level system description language as the single user input medium. This takes the form of a net list of bit-serial 'operators' that represent a flow graph of the system to be implemented. The task of the compiler is to compose a complete chip layout by assembling and placing these operators, and then routing the network.

FIRST maintains a restricted library of primitive operators (multiply, add, delay, etc.) that includes all of the basic processes needed to build a wide range of signal processing systems. It also allows the user

ESSCIRC '82 Digest of Technical Papers
Brussels, September 1982. pp.215-218

APPENDIX IV

LIST OF PUBLICATIONS

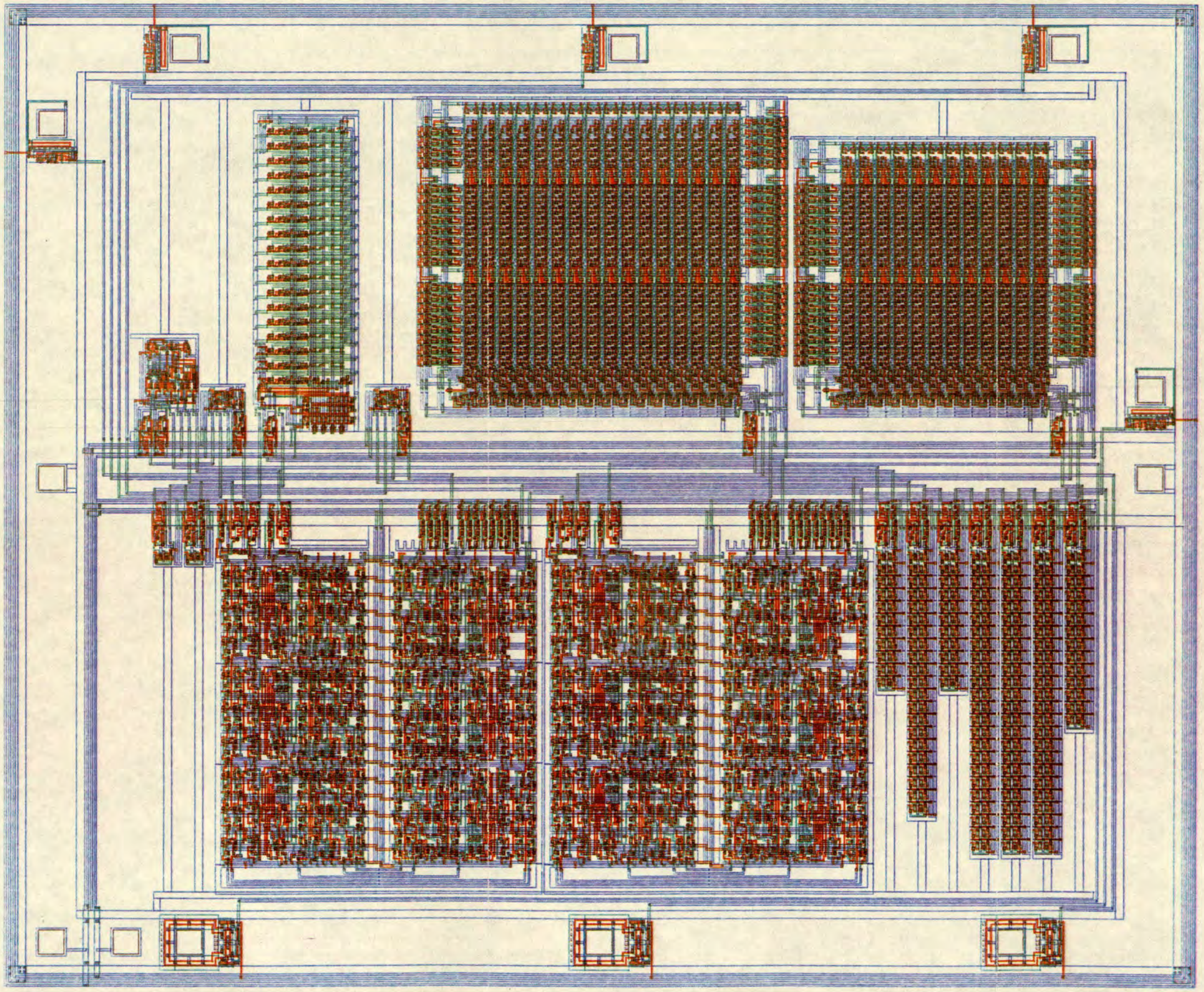
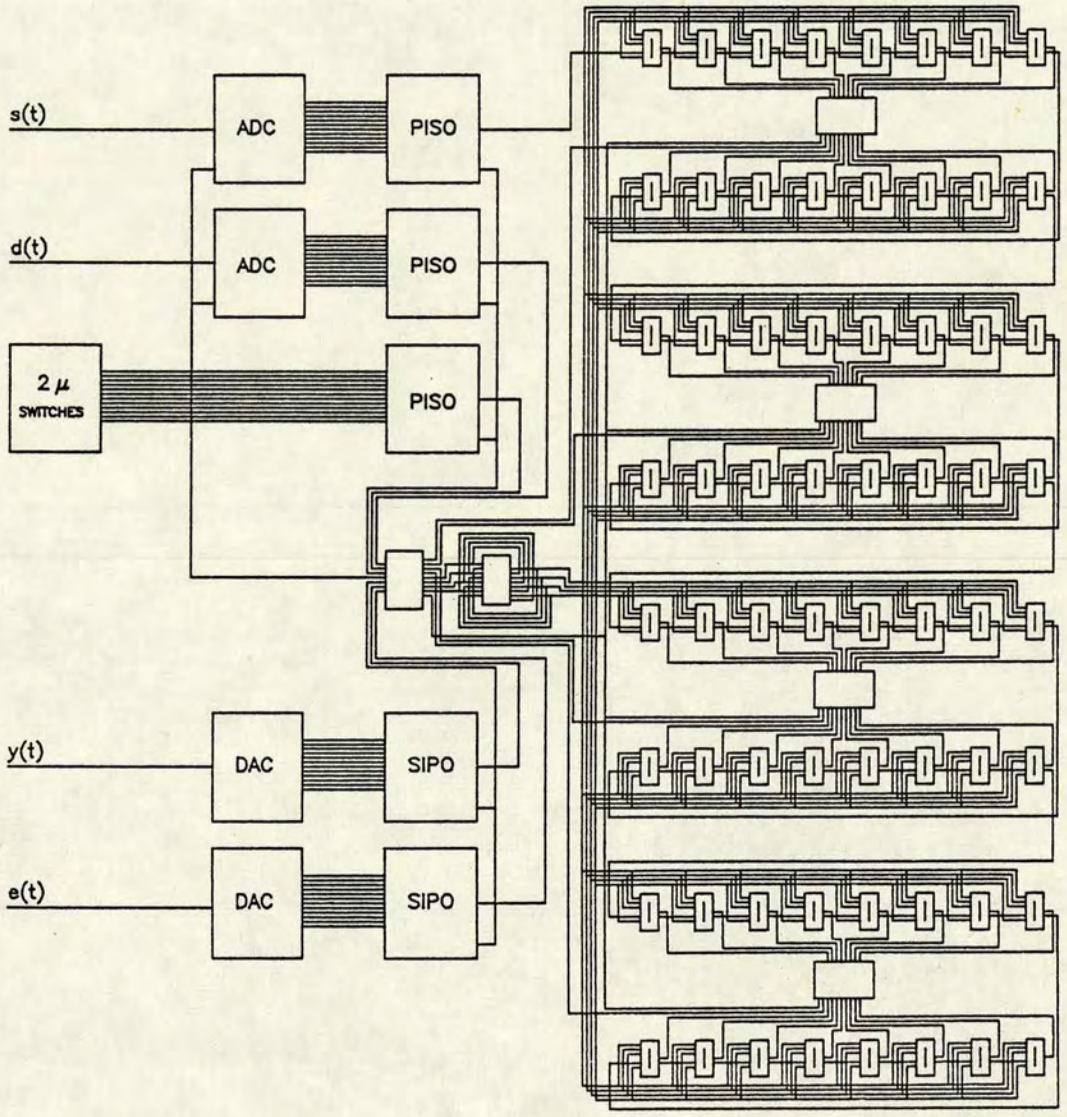


Figure 19

LMS Adaptive Echo Canceller – Wiring Schematic



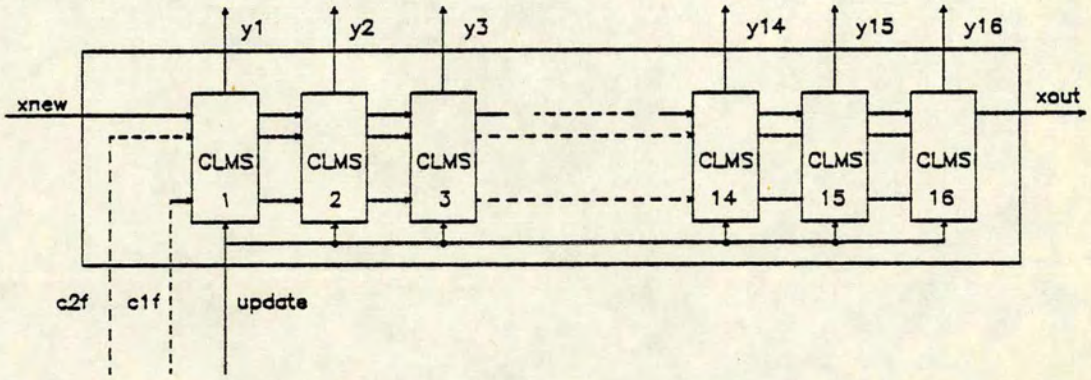


Figure 16: SUBSYSTEM ZFIR

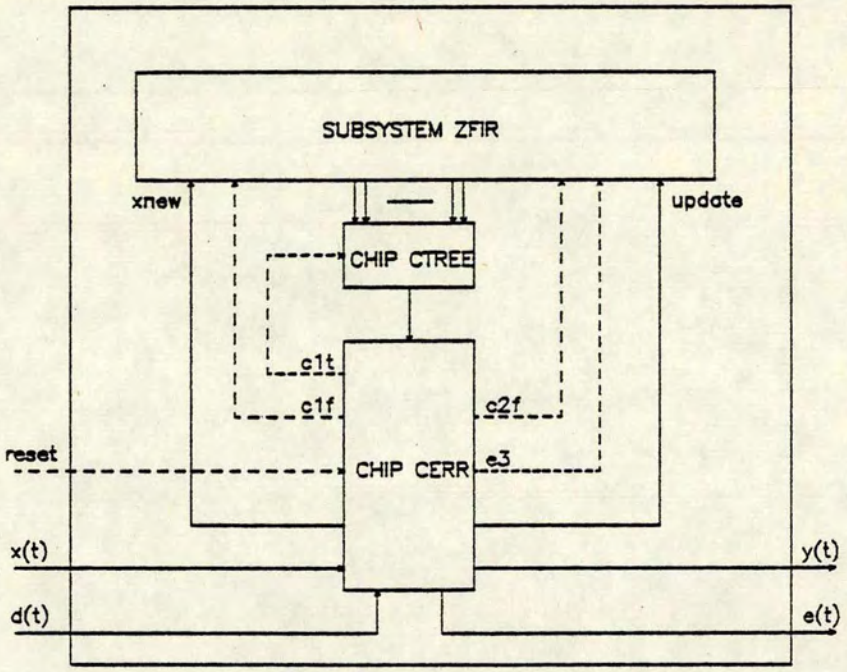


Figure 17: SYSTEM SECHOCANC

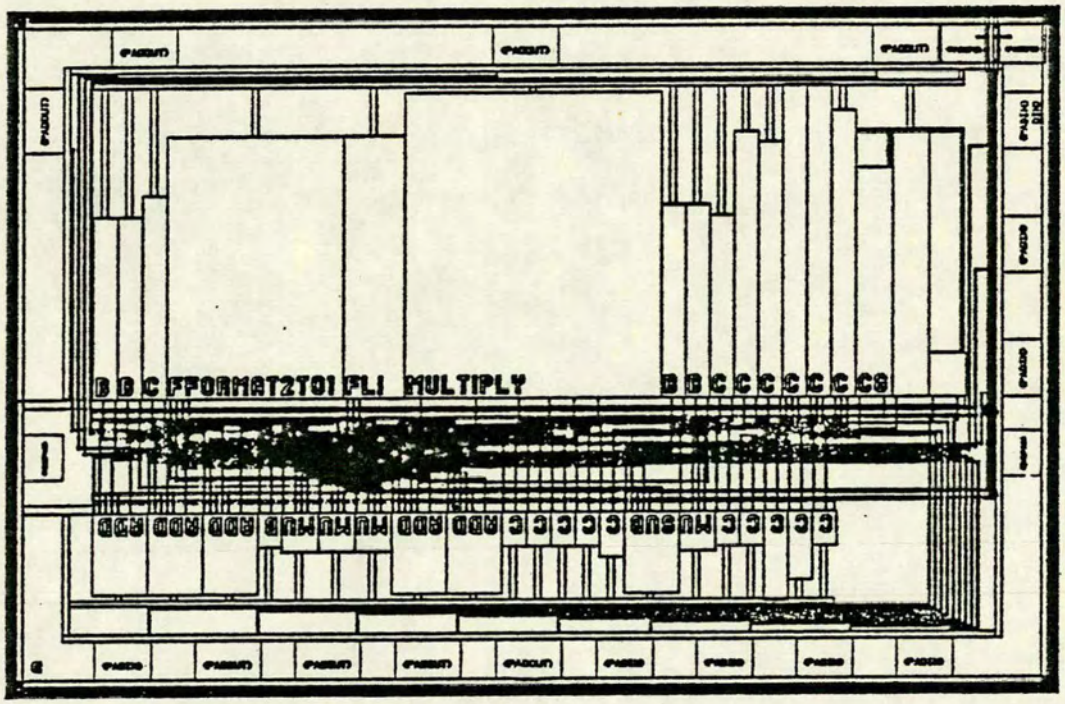


Figure 15: CERR floor plan

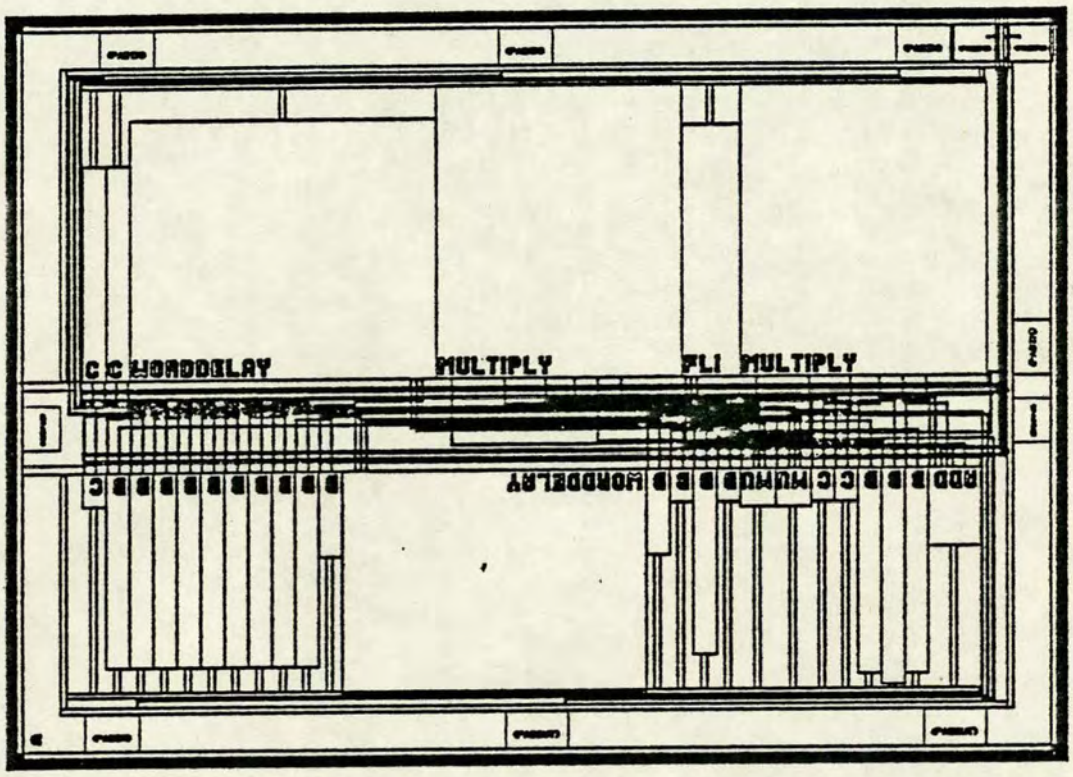


Figure 13: CLMS floor plan

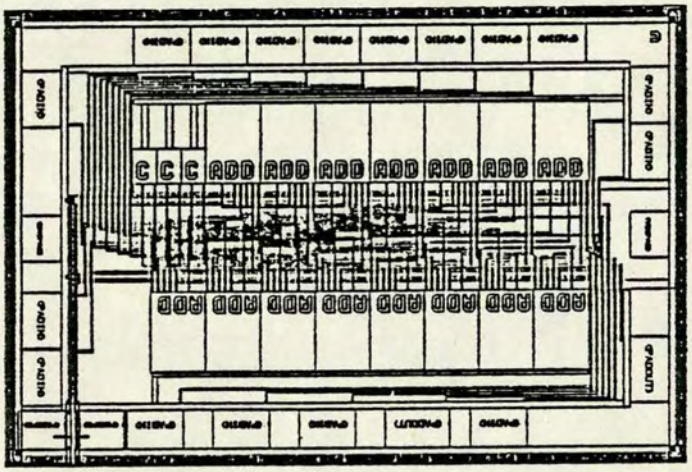


Figure 14: CTREE floor plan

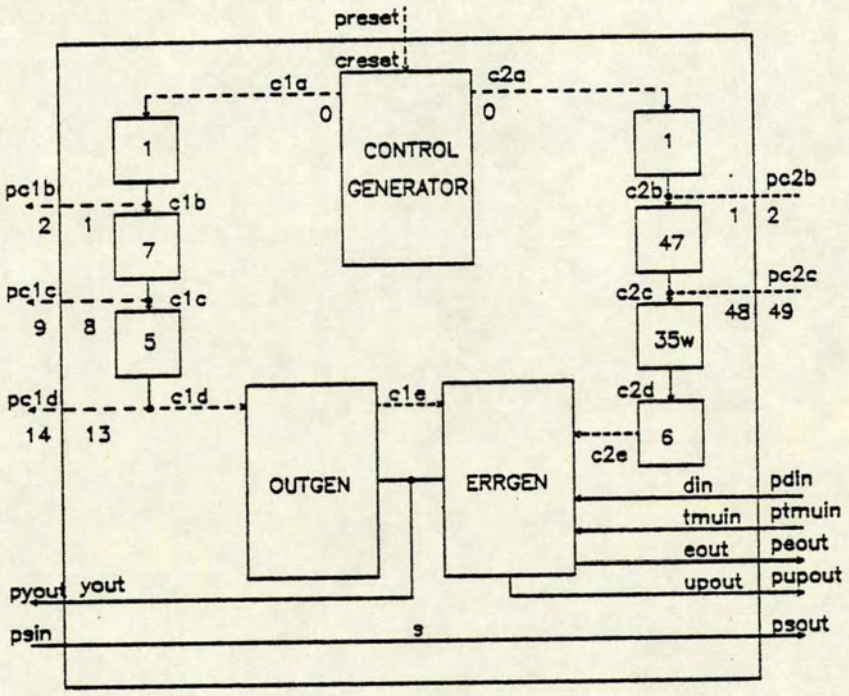


Figure 12: CHIP CERR

```
CHIP CERR2 (preset -> pc1b, pc1c, pc2b, pe3) psin, pin1, pin2, pin3,
           pin4, pdin, ptmuin -> peout, psout, pupout, pyout

   SIGNAL s, din, tmuin, upout, eout, in1, in2, in3, in4, yout
   CONTROL c1a, c1b, c1c, c1d, c1e, c2a, c2b, c2c, c2d, c2e,
           e3, c2b1, creset, c2c1

   CONSTANT sw1 = 18
   CONSTANT c = 15

   PADIN (preset -> creset) pdin, ptmuin, psin -> din, tmuin, s
   PADOUT (c1b, c1c, c2b, e3 -> pc1b, pc1c,
          pc2b, pe3) eout, upout, s -> peout, pupout, psout
   PADORDER VDD, preset, pc1b, pc1c, pc2b, pe3, pin1, pin2, pin3, pin4,
            GND, psin, pdin, ptmuin, CLOCK, peout, pupout, psout, pyout


   PADIN pin1, pin2, pin3, pin4 -> in1, in2, in3, in4
   PADOUT yout -> pyout

   OUTGEN [sw1] (c1d, c2c -> c1e) in1, in2, in3, in4 -> yout
   ERRGEN [sw1,c] (c1e, c2e ->) yout, din, tmuin -> eout, upout

   CBITDELAY [1] (c1a -> c1b)
   CBITDELAY [7] (c1b -> c1c)
   CBITDELAY [5] (c1c -> c1d)
   CBITDELAY [1] (c2a -> c2b)
   CBITDELAY [24] (c2b -> c2b1)
   CBITDELAY [23] (c2b1 -> c2c)
   CWORDDELAY [12,0] (c1a, c2c -> c2c1)
   CWORDDELAY [12,0] (c1a, c2c1 -> c2d)
   CWORDDELAY [11,6] (c1a, c2d -> c2e)

   CONTROLGENERATOR (creset -> NC, c1a, c2a, e3)
      CYCLE [18]
      CYCLE [40]
      EVENT
   ENDCONTROLGENERATOR

END
endofprogram
```

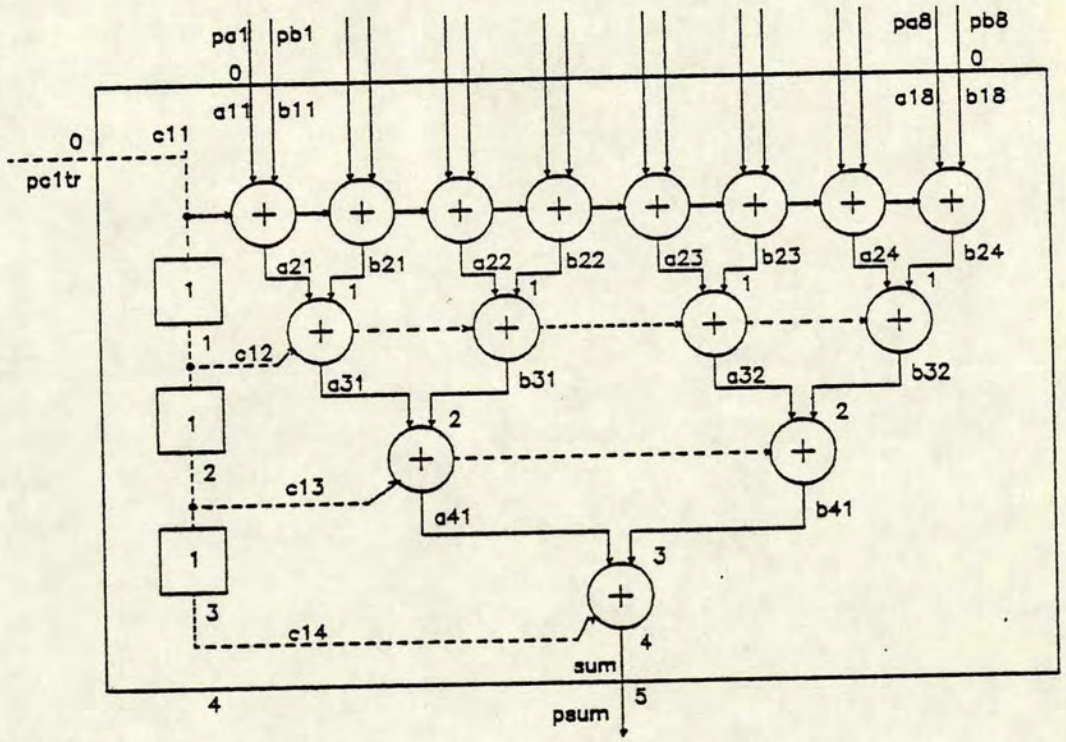


Figure 11: CHIP CTREE

```
CHIP CTREE1 (pc1i -> pc1o) pa1, pa2, pa3, pa4, pa5, pa6,
           pa7, pa8, pb1, pb2, pb3, pb4, pb5, pb6, pb7, pb8 -> psum

   SIGNAL a11, a12, a13, a14, a15, a16, a17, a18,
          b11, b12, b13, b14, b15, b16, b17, b18,
          a21, a22, a23, a24, b21, b22, b23, b24,
          a31, a32, b31, b32, a41, b41, sum
   CONTROL c11, c12, c13, c14

   PADIN (pc1i -> c11) pa1, pa2, pa3, pa4, pa5, pa6, pa7, pa8, pb1, pb2,
         pb3, pb4, pb5, pb6, pb7, pb8 -> a11, a12, a13, a14, a15, a16, a17,
         a18, b11, b12, b13, b14, b15, b16, b17, b18
   PADOUT (c14 -> pc1o) sum -> psum

   PADORDER VDD, pa1, pb1, pa2, pb2, pa3, pb3, pa4,
            pb4, pa5, pb5, pa6, GND, pb6, pa7,
            CLOCK, pb7, pa8, pb8, psum, pc1i, pc1o

   ADD [1,0,0,0] (c11) a11, b11, GND -> a21, NC
   ADD [1,0,0,0] (c11) a13, b13, GND -> a22, NC
   ADD [1,0,0,0] (c11) a15, b15, GND -> a23, NC
   ADD [1,0,0,0] (c11) a17, b17, GND -> a24, NC
   ADD [1,0,0,0] (c11) a12, b12, GND -> b21, NC
   ADD [1,0,0,0] (c11) a14, b14, GND -> b22, NC
   ADD [1,0,0,0] (c11) a16, b16, GND -> b23, NC
   ADD [1,0,0,0] (c11) a18, b18, GND -> b24, NC
   ADD [1,0,0,0] (c12) a21, b21, GND -> a31, NC
   ADD [1,0,0,0] (c12) a23, b23, GND -> a32, NC
   ADD [1,0,0,0] (c12) a22, b22, GND -> b31, NC
   ADD [1,0,0,0] (c12) a24, b24, GND -> b32, NC
   ADD [1,0,0,0] (c13) a31, b31, GND -> a41, NC
   ADD [1,0,0,0] (c13) a32, b32, GND -> b41, NC
   ADD [1,0,0,0] (c14) a41, b41, GND -> sum, NC

   CBITDELAY [1] (c11 -> c12)
   CBITDELAY [1] (c12 -> c13)
   CBITDELAY [1] (c13 -> c14)

END
```

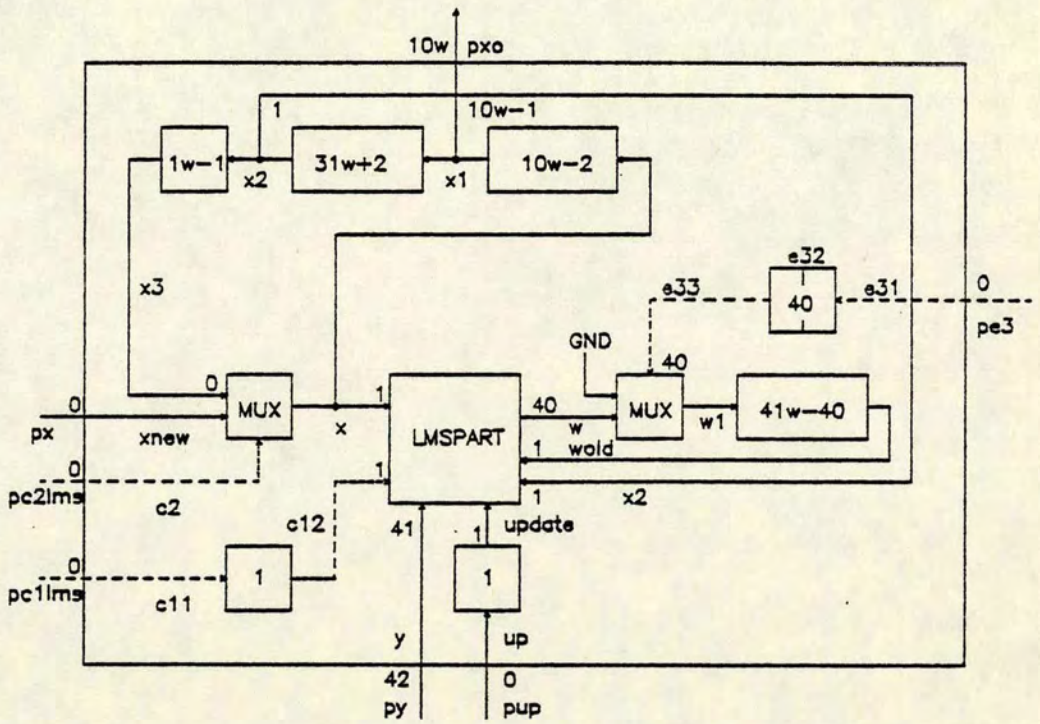


Figure 10: CHIP CLMS

```
CHIP CLMS1 (pc1,pc2,pe3) px, pup -> pxo, py

   SIGNAL xnew, x1, x2, x3, x, xold, w, w1, wold, up, update, y,
          xx1, xx2, xx3, xx4, xx5, xx6, xx7, xx8, xx9, xy1, w11
   CONTROL c11, c12, c2, e31, e32, e33

   CONSTANT sw1=18

   PADORDER VDD, px, pxo, py, GND, pup, CLOCK, pe3,
            pc1, pc2

   PADIN (pc1,pc2,pe3 -> c11,c2,e31) px, pup -> xnew, up
   PADOUT x1, y -> pxo, py

   CBITDELAY [1] (c11 -> c12)
   CBITDELAY [20] (e31 -> e32)
   CBITDELAY [20] (e32 -> e33)

   BITDELAY [sw1] x -> xx1
   BITDELAY [sw1] xx1 -> xx2
   BITDELAY [sw1] xx2 -> xx3
   BITDELAY [sw1] xx3 -> xx4
   BITDELAY [sw1] xx4 -> xx5
   BITDELAY [sw1] xx5 -> xx6
   BITDELAY [sw1] xx6 -> xx7
   BITDELAY [sw1] xx7 -> xx8
   BITDELAY [sw1] xx8 -> xx9
   BITDELAY [7] xx9 -> x1
   WORDDELAY [29,sw1,1] (c11) x1 -> xy1
   BITDELAY [7] xy1 -> x2
   WORDDELAY [37,sw1,1] (c11) w1 -> w11
   BITDELAY [2] w11 -> wold
   BITDELAY [sw1-1] x2 -> x3
   BITDELAY [1] up -> update

   MULTIPLEX [1,0,0] (e33) w, GND -> w1
   MULTIPLEX [1,0,0] (c2) x3, xnew -> x

   LMSPART1 [sw1] (c12) x, update, x2, wold -> y, w

END
```

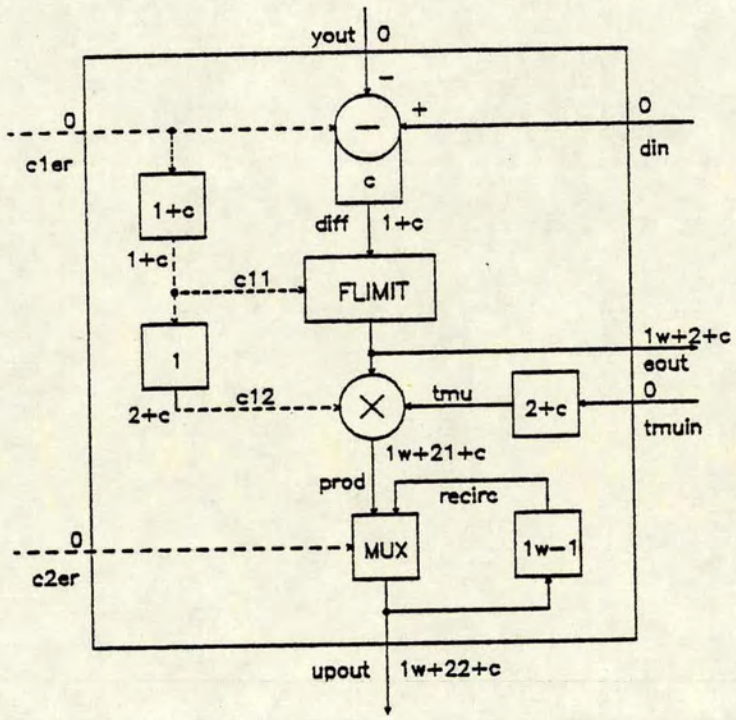


Figure 9: OPERATOR ERRGEN

```
OPERATOR ERRGEN [sw1,c] (c1er, c2er, ->) yin, din,
         tmuin -> eout, upout

   SIGNAL diff, prod, tmu, recirc
   CONTROL c11, c12

   SUBTRACT [1,0,0,c] (c1er) din, yin, GND -> diff, NC
   FLIMIT [sw1,1,0] (c11) diff -> eout
   MULTIPLY [1,12,0,0] (c12 -> NC) eout, tmu -> prod, NC
   MULTIPLEX [1,0,0] (c2er) prod, recirc -> upout
   BITDELAY [sw1-1] upout -> recirc
   BITDELAY [2+c] tmuin -> tmu

   CBITDELAY [1] (c11 -> c12)
   CBITDELAY [1+c] (c1er -> c11)

END
```

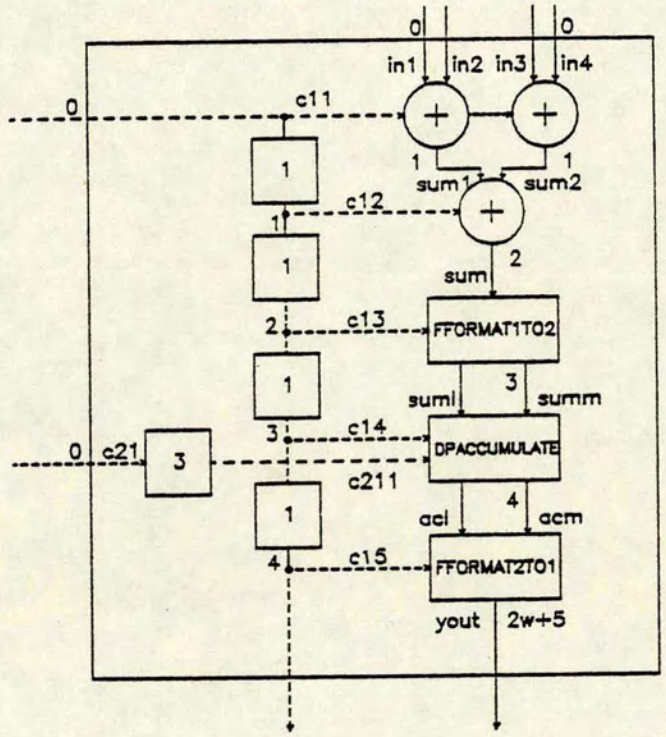


Figure 8: OPERATOR OUTGEN

```
OPERATOR OUTGEN [sw1] (c1i, c2i, -> c1o) in1, in2, in3, in4 -> yout

   SIGNAL sum1, sum2, sum, suml, summ, acl, acm
   CONTROL c12, c13, c14, c22, c211

   ADD [1,0,0,0] (c1i) in1, in2, GND -> sum1, NC
   ADD [1,0,0,0] (c1i) in3, in4, GND -> sum2, NC
   ADD [1,0,0,0] (c12) sum1, sum2, GND -> sum, NC
   SIGNEXTEND (c13) sum -> suml, summ
   DPACCUMULATE [sw1] (c14, c211 -> ) suml, summ -> acl, acm
   FFORMAT2TO1 [sw1,0,0,0,0] (c1o) acl, acm -> yout

   CBITDELAY [1] (c1i -> c12)
   CBITDELAY [1] (c13 -> c14)
   CBITDELAY [1] (c12 -> c13)
   CBITDELAY [1] (c14 -> c1o)
   CBITDELAY [3] (c2i -> c211)

END
```

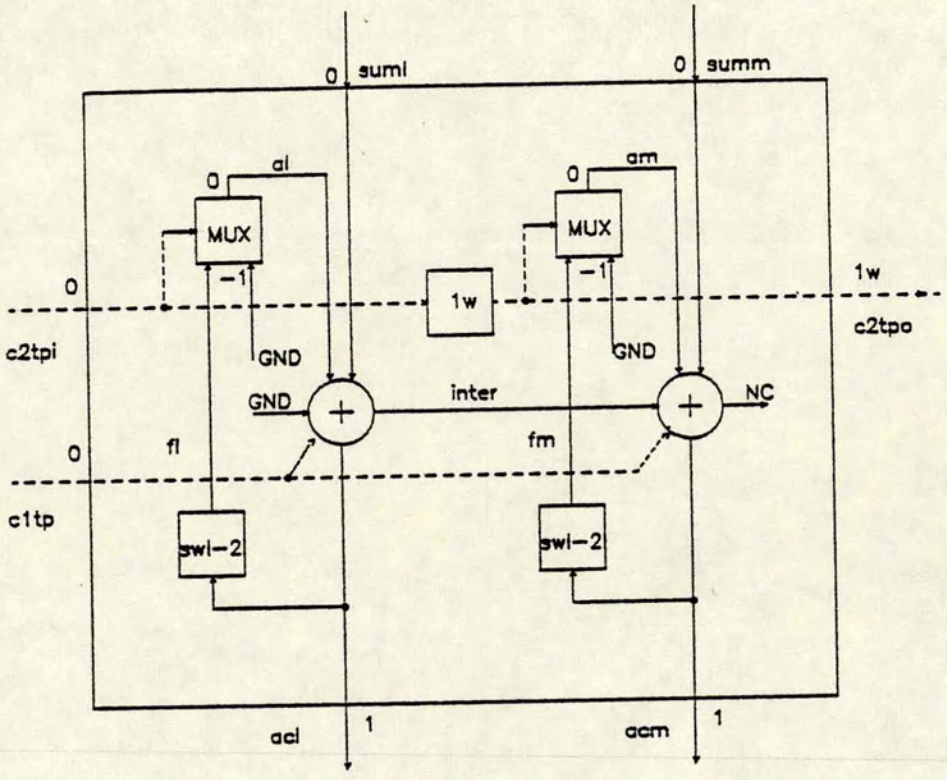


Figure 7: OPERATOR DPACCUMULATE

```
OPERATOR DPACCUMULATE [swl] (c1dp, c2dpi -> ) suml,
                            summ -> acl, acm

   SIGNAL al, am, int, fl, fm
   CONTROL c22

   MULTIPLEX [1,0,0] (c2dpi) fl, GND -> al
   MULTIPLEX [1,0,0] (c22) fm, GND -> am
   ADD [1,0,0,0] (c1dp) al, suml, GND -> acl, int
   ADD [1,0,0,0] (c1dp) am, summ, int -> acm, NC
   BITDELAY [swl-2] acl -> fl
   BITDELAY [swl-2] acm -> fm

   CBITDELAY [swl] (c2dpi -> c22)

END
```

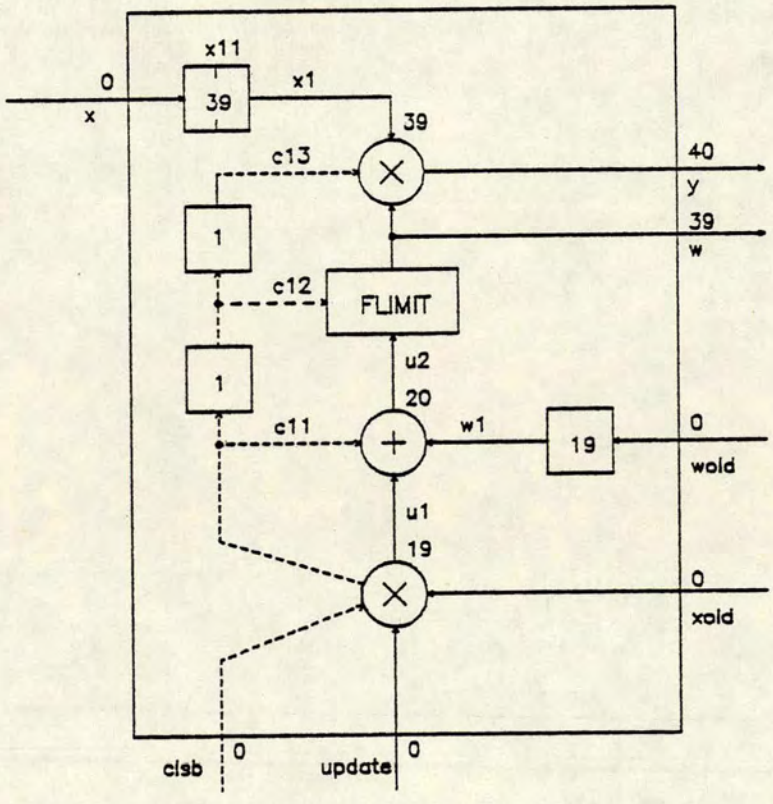


Figure 6: OPERATOR LMSPART

```
OPERATOR LMSPART1 [sw1] (clsb) x, update, xold, wold -> y, w

   SIGNAL x1, u1, u2 ,wold1, x11
   CONTROL c11, c12, c13

   CBITDELAY [1] (c11 -> c12)
   CBITDELAY [1] (c12 -> c13)

   BITDELAY [19] x -> x11
   BITDELAY [20] x11 -> x1
   BITDELAY [19] wold -> wold1

   MULTIPLY [1,12,0,0] (c13 -> NC) w, x1 -> y, NC
   FLIMIT [sw1,1,0] (c12) u2 -> w
   ADD [1,0,0,0] (c11) u1, wold1, GND -> u2, NC
   MULTIPLY [1,12,0,0] (clsb -> c11) update, xold -> u1, NC

END
```

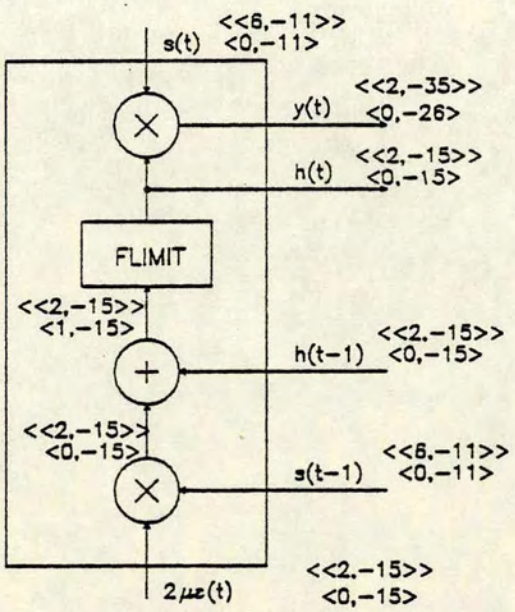


Figure 4: single LMS processor

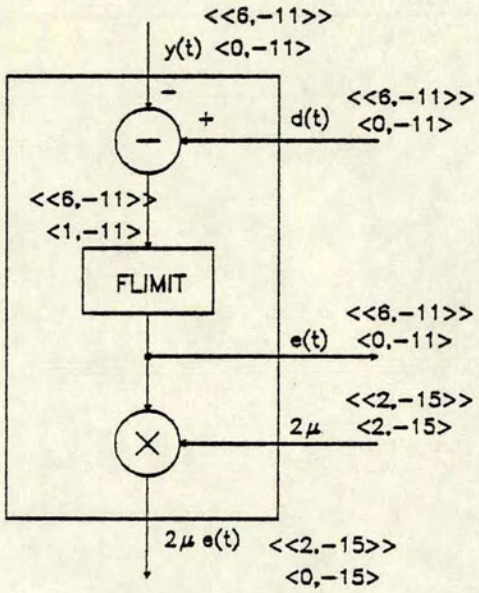


Figure 5: error generator

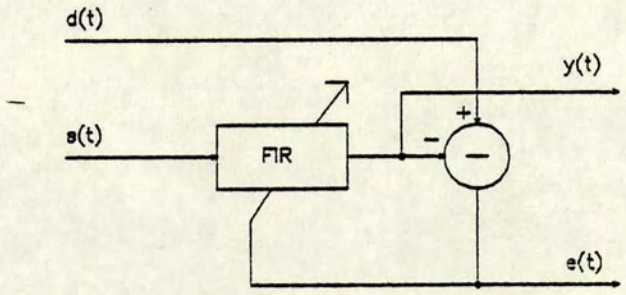


Figure 1: Widrow LMS adaptive filter

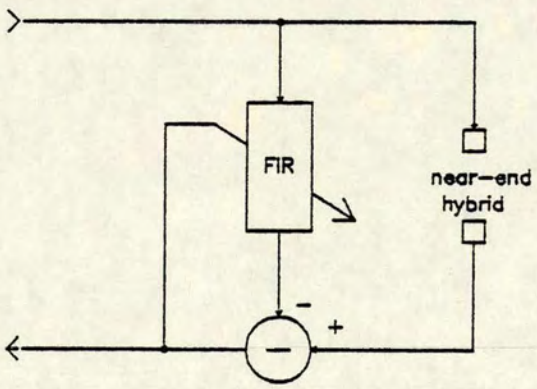


Figure 2: telephone line echo canceller

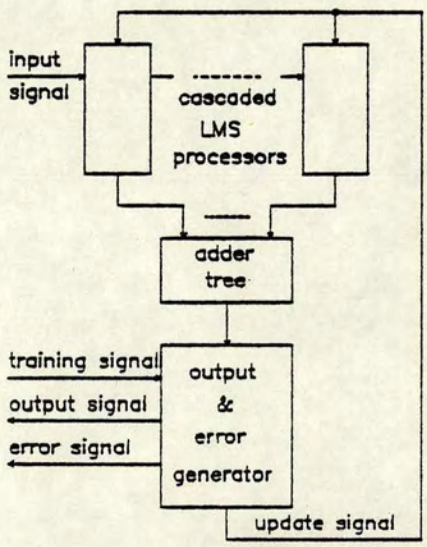


Figure 3: initial system plan

References

1. B.Widrow et. al., "Adaptive noise cancelling: principles and applications," Proc. IEEE, Vol. 63, pp. 1692-1716 (December 1975).

2. D.R.Morgan, "Adaptive multipath cancellation for digital data communications," IEEE Trans. Comms., Vol. COM-26, (9) pp. 1380-1390 (September 1978).

3. P.B.Denyer, D.Renshaw, and N.Bergmann, "A silicon compiler for VLSI signal processors," Proc ESSCIRC '82, pp. 215-218 (September 1982).

3. ERR - a block in which OUTGEN sums and formats the outputs of 4 TREE operators (i.e. 2048 filter points), accumulates in double-precision, and re-formats to single precision. The output is passed to ERRGEN, which performs subtraction to produce the error sample, limits, and multiplies by 2μ.

The system is now partitioned into feasible chips. The three chip designs (Figs. 10 - 12), corresponding to the three operators listed above, take the names CLMS, CTREE and CERR. The compensating delay c, and the control timing are adjusted to accommodate the bit-delays introduced across chip boundaries. c is calculated to be 15 bits. The control generator is placed on CERR.

The system flow diagram, up to chip level, is now complete, and may be entered to the silicon compiler for chip composition. Dimensions of the completed chips are as follows:

chip	width (λ)	height (λ)
CLMS	2531	1807
CTREE	1565	1050
CERR	2412	1582

Chip floor plans are given in Figures 13 - 15. Unfortunately it is the largest chip which must be repeated many times within the system, although we feel that even this device is not excessively large.

Extending the hierarchy to the remaining levels (subsystems and systems,) the outstanding block diagrams were produced. Fig.16 shows the subsystem ZFIR, containing 16 cascaded CLMS chips, and Fig.17 the entire system, SECHOCANC. However FIRST language files have not at this stage been produced for these levels of the design.

The system of Fig.17 represents a 512-point filter. Should the full length of 2048 be required, 4 ZFIRs, each of which feeds a CTREE, may be cascaded. Operator OUTGEN gives CERR the facility to accept inputs from up to 4 CTREE chips.

7. System simulation

At the time of preparation of this report, the system had not yet been simulated.

8. Conclusions

An echo canceller has been described which is cascadable, in blocks of 512, to length 2048 filter points. A set of three LSI chips has been specified, designed and laid out in approximately three man-weeks, by a first-time user of the FIRST silicon compiler. It is anticipated that another three man-weeks should see the project through to completion.

critical path length form this figure. The critical path length is the length of the algorithm recursion loop, i.e. the number of word delays between the appearance of the final multiplexed signal sample on the LMS block and the appearance of the error-update sample. This critical path length must be accommodated at the end of each filter output cycle, since the next cycle may not commence until the error has been computed and returned to the LMS blocks. As each primitive, operator etc. in the path has an intrinsic delay, this figure may be easily estimated.

The critical path length was estimated at around 8 words, giving a multiplexing potential of 47. However, in order to reduce the multiplexing memory requirement (and hence the chip size), and also relax the 8 MHz clock rate (which is near the process limit), the multiplexing level was chosen to be 32. This leads to a revised system clock rate of (32 + 8) x 8 kHz x 18, = 5.76 MHz.

As the critical path length is not an integer number of system wordlengths, a block of compensating delay must be introduced into the critical path, to give it an integer length. Whilst this may be accomplished by simply inserting a bit-delay block (the delay being parameterised for the present), a more elegant solution is to use the compiler's facility for entering arbitrary delay after such primitives as ADD and SUBTRACT, thereby saving on chip floorspace. The subtractor which produces e(t) from d(t) and y(t) is ideally suited for introduction of this compensating delay.

Having completed the system description, and estimated the partitioning into chips, the next step was to create source files for the FIRST silicon compiler.

6. System implementation using FIRST

Having designed the system from the top down, it could now be implemented from the bottom up. The first step was to create the arithmetic operators. These were identified as follows:

1. LMSPART - an LMS processor block

2. DPACCUMULATE - a double-precision accumulator

3. OUTGEN - a block which produces and formats y(t) by accumulating the outputs of all LMSPART blocks, and

4. ERRGEN - a block which produces 2μe using y(t), d(t) and 2μ.

Figs. 6 - 9 depict these operators as computation engines constructed from FIRST primitives. The FIRST source file is displayed in each case. As the primitive FFORMAT2TO1 had not at this stage been entered to the compiler, it was substituted by the operator SIGNEXTEND, which consisted of a bit delay and a multiplexer.

The next step was to construct the following three operators:

1. LMS - an LMSPART operator multiplexed to produce 32 filter points, with associated multiplexing memory

2. TREE - an adder tree capable of summing the outputs of 16 LMS operators, i.e. 512 filter points, and

be multiplexed over as many time divisions as system constraints would allow. It was assumed that a single LMS processor plus the multiplexing memory would fit on to a single chip. The level of multiplexing attainable was then a function of tolerable chip dimensions, processor latency, system clock rate and system bandwidth. The error generator in turn fed the LMS blocks, thus completing the system loop. Fig.3 is the initial chip plan of the system.

5. System description

Starting from the algorithmic equations (eqns. 1 - 3), block diagrams were produced, detailing the construction of a single LMS processor block, and the error-update block. Figs. 4 and 5 show these blocks. Initially it was thought that the multiplier used in the filter output convolution should produce a double-precision output, which should be accumulated in triple-precision to allow for word growth. However, the nature of echo-cancellation is such that the filter produces a noisy estimate of the echo path, and it was decided that a single-precision rounded-output multiplier, accumulated to double-precision, would suffice.

The next step was to determine the significance and format of words at each point in the system. The longest desired significance in the system is the 16-bit tap weight. Assuming no further restrictions on wordlength occur (which turns out to be true) then the system wordlength may be set at 2 guard bits above 16, i.e. swl = 18. Thus the word rate for the echo canceller engines is 8 MHz divided by 18 bits, i.e. 444.4 kHz.

Figs.4 and 5 display the significance (single arrows) and format (double arrows) of words at important points of the system. Format and significance in the adder tree is not treated, as the double precision nature of this structure precludes overflow and other undesirable phenomena.

The system can be said to work in three distinct, cycling modes:

1. Operation, i.e. performing convolution to produce an output sample (cf. eqn. 1)

2. Error-update generation, i.e. subtraction of the formatted output sample from the training signal (cf. eqn. 2) then multiplying by 2μ, and

3. Adaptation, i.e. multiplication of $2\mu e$ by each signal sample and accumulation in the tap weight registers (cf. eqn. 3)

Due to the pipelining inherent in bit-serial systems, mode 3 may be executed in parallel with mode 1. Mode 2, however, cannot proceed until mode 1 starts to produce the final accumulated output word, and mode 3 must await the appearance of the result of mode 2 before commencing. Thus mode 2, whilst computationally trivial in comparison to the other modes, provides an unavoidable system bottleneck during which the other modes must idle.

The level of multiplexing in the LMS blocks may be decided at this stage. The full multiplexing potential of the system, i.e. ignoring idles and latencies for the present, is simply the ratio of system word rate to system sample rate. In this case, the full potential is 444.4 / 8 , i.e. 55.55. To obtain the actual potential, we must subtract the

$$e^2(t) = d^2(t) - 2d(t)\mathbf{S}^T\mathbf{H} + \mathbf{H}^T\mathbf{S}\mathbf{S}^T\mathbf{H}$$

and differentiating w.r.t. H gives

$$\frac{\partial e^2(t)}{\partial \mathbf{H}} = -2d(t)\mathbf{S} + 2\mathbf{S}\mathbf{S}^T\mathbf{H}$$

Eqns.1 and 2 give

$$\frac{\partial e^2(t)}{\partial \mathbf{H}} = -2\mathbf{S}e(t)$$

The updates to the tap weights are performed by subtracting these correlation estimates (multiplied by a convergence factor μ) from each tap weight. The size of μ determines the accuracy and stability of the final solution, and the speed of convergence to that solution. The maximum bound on μ is governed by input signal statistics [1]. The updating process may be expressed as

$$\mathbf{H}' = \mathbf{H} - \mu\frac{\partial e^2(t)}{\partial \mathbf{H}}$$

i.e.

$$\mathbf{H}' = \mathbf{H} + 2\mu\mathbf{S}e(t) \qquad (3)$$

Eqns. 1 - 3 represent the form of the algorithm to be implemented in the echo-canceller under design.

4. System specification and initial chip plan

From a designer's point of view, the echo canceller is manifestly a straightforward adaptive filter. However, due to the nature of the echo path in long-distance telephone networks, the canceller may be required to exhibit a long impulse response, requiring two thousand or more filter points to achieve accurate modelling. This involves the designer in problems of word growth over the filter accumulation, and necessitates partitioning of the system into identical cascadable blocks.

The echo-canceller chip set of this case study was specified as follows:

1. Filter bandwidth = 4 kHz (i.e. sample rate = 8 kHz)
2. Filter length = 256 in basic form, up to 2048 in full form
3. Converter resolution = 12 bits
4. Tap weight resolution = 16 bits

The implementation of the canceller was undertaken as a case study in system design using the FIRST silicon compiler [3]. The compiler forms a chip plan (using bit-serial architecture) from a high level list of connected primitives, operators etc., generated by the user. The target process for the compiler is currently the Edinburgh Microfabrication Facility's 6-micron nMOS process. The compiler itself specified a system clock rate of 8 MHz.

The system was initially proposed as a set of identical cascaded LMS processor blocks feeding an adder tree which in turn fed an error-generating chip containing the control generator. Each LMS block was to

1. Introduction

Recent years have seen the emergence of many digital adaptive filter designs, a great proportion of which use variations of the Widrow LMS algorithm [1]. The functional block diagram for such a filter is shown in Fig.1. With the advances in fabrication technologies and computer-assisted design techniques, more and more of these systems are being realised as VLSI chips or chip sets.

A particular application area for adaptive filters is echo cancellation in telephone networks. Due to various reasons [2], the received signal is corrupted by delayed versions of the transmitted signal. The solution is to adaptively model the echo path, and subtract the filtered received signal from the transmitted signal. If this process is carried out at both ends of the line, echo-free communication may be enjoyed. Fig.2 shows the usual configuration for a telephone line echo canceller.

2. List of symbols

The following symbols are used:

N = number of filter points
n = variable scanning the set of N filter points
t = time
$s(t-n)$ = the signal at filter point n, at time t
$h(n,t)$ = the tap weight at filter point n, at time t
$d(t)$ = the training or conditioning signal at time t
$y(t)$ = the filter output signal at time t
$e(t)$ = the filter error signal at time t
$\mathbf{S}$ = the vector of signal samples in the filter at time t
$\mathbf{H}$ = the vector of filter tap weights at time t
$\mathbf{H}'$ = the vector of filter tap weights at time t+1
$\mathbf{S}^T$ = the transposed signal vector $\mathbf{S}$

3. Theoretical background

A transversal, or finite impulse response (FIR) filter produces its output sample by convolving its signal and tap weight vectors, i.e.

$$y(t) = \mathbf{S}^T\mathbf{H} \tag{1}$$

The Widrow LMS algorithm operates as follows. Firstly the output sample $y(t)$ is subtracted form a training signal $d(t)$ to produce the error sample $e(t)$, i.e.

$$e(t) = d(t) - y(t) \tag{2}$$

The algorithm forms an estimate of the correlation between the error sample and the signal sample at each tap in the filter, and updates each tap weight in such a direction as to reduce this correlation. The filter is said to have converged when the error signal and the signal vector are orthogonal.

A useful cost function, describing the distance of the tap weight vector from the optimal solution (Wiener vector) [1], is mean-square error (MSE). The object of the algorithm is then to minimise the MSE. The Widrow LMS algorithm forms a crude estimate of the MSE by squaring a single error sample, and minimises this by gradient methods. Substituting eqn.(1) in eqn.(2) gives

ABSTRACT

A chip set is described for echo cancellation on telephone lines. Use is made of the FIRST silicon compiler, which drastically reduces design time. A set of three chips is built from a collection of FIRST operators, both chips and operators being described in detail. Some insight into the design process is given. Finally floor plans of the three IC designs are presented, along with projections for the completion of the project.

Case Studies in System Design with a Silicon Compiler

An LMS Adaptive Transversal Filter for Speech Echo Cancellation

Report No. ISG/83/11

INTERIM REPORT

January, 1983

Stewart G. Smith
David Renshaw
Peter B. Denyer

Integrated Systems Group, Department of Electrical Engineering,
University of Edinburgh, The King's Buildings, Edinburgh, UK.

test chip (mentioned in connection with testing the code defining the behaviour of the primitive). This compiled chip is then fabricated and tested to demonstrate equivalence of function, and to verify performance. Failure on this testing necessitates cell redesign. Once a primitive has been verified it can be accepted into the cell library.

declarations allocates a constant associated with design style. This is used to orient and place the final composed primitive. The second grouping of lines retrieves the primitive parameter values. The third group of lines is concerned with generating a unique name for the primitive; this is based on the primitive name and parameter values. Next a test is made to determine whether the primitive with these parameter values has already been generated. If this has not been done then a symbol with the appropriate name is generated. The code for this is contained between the line:

```
symbol(ttname);
```

and the line:

```
endsymbol;
```

The fifth, sixth and seventh groups of lines of code are between these lines. The fifth group places the appropriate input connections, or predelay buffers conditionally selecting on parameter value. The sixth group of lines places the output buffers. The seventh group of lines is the main composition and selects appropriate cells depending on the parameter value, in this case the latency, and places them. This portion of code involves conditional selection and placement as well as limited repetition, dependent on parameter value. Of the two remaining groups of lines the eighth determines the dimensions of the finished primitive and returns this information to the main placement programs and the ninth does the same for input and output port positions.

Final verification of a primitive is achieved by compiling the

```
        }
        endsymbol();

    }

    if(n == 1) {
        *f_height=186;
        *f_width= 126;
    }
    else {
        if(n > 1 && n < 5)
            *f_height=224;
        else {
            q=(n-5)/4;
            r=(n-5)-(4*q);
            i=8;
            if(r == 0 || r == 1)
                *f_height = 216;
            else
                *f_height = 234;
            while(i < n) {
                *f_height += 36;
                i += 4;
            }
            if(n == 2*(n/2))
                *f_height += 32;
            else
                *f_height += 14;
        }
        *f_width = 140;
    }

    f_places[1 + 1] = 122;
    f_places[2 + 1] = 80;
    f_places[3 + 1] = 94;
    f_places[4 + 1] = 38;
    f_places[5 + 1] = 66;
    f_places[6 + 1] = 108;
    if(n > 1) {
        for(i = 1;i <= 6;++i)
            f_places[i + 1] += 14;
    }

}
```

Table III.5

Again the lines of code have been grouped, in order to show the overall structure, which is as follows. The first line after the

```
draw("ADDDEL",x,y);
x=x+136;
if(n == 2)
     draw("ASDELA",x,y);
if(n == 3)
     draw("ASDELB",x,y);
if(n == 4)
     draw("ASDELC",x,y);
else {
     if(n > 4) {
          q=(n-5)/4;r=(n-5)-(4*q);
          i=8;
          if(r == 0 || r == 1) {
               draw("ASDELD",x,y);
               x=x+38;
          }
          else {
               draw("ASDELG",x,y);
               x=x+56;
          }
          while(i < n) {
               draw("ASDELH",x,y);
               y=127;
               if(r == 0 || r == 1) {
                    draw("ASDELI",x,y);
                    x=x+36 ; y=0;
               }
               else {
                    draw("ASDELJ",x,y);
                    x=x+36 ; y=0;
               }
               i += 4;
          }
          if(n == 2*(n/2)) {
               draw("ASDELF",x,y);
               y=127;
               if(r == 0 || r == 1)
                    draw("ASDELM",x,y);
               else
                    draw("ASDELN",x,y);
          }
          else {
               draw("ASDELE",x,y);
               y=127;
               if(r == 0 || r == 1)
                    draw("ASDELK",x,y);
               else
                    draw("ASDELL",x,y);
          }
     }
}
```

```
add()
{
int i,x,y;
int q,r,n,inA,inB,inC;

     designer = RENSHAW;

     n = f_plist[1];
     inA = f_plist[2];
     inB = f_plist[3];
     inC = f_plist[4];

     s(ans1,1);
     s(ans2,2);
     s(ans3,3);
     sprintf(ttname,"%s[%s,%s,%s]",ssname,ans1,ans2,ans3);

     if(!symbol_exists(ttname) && full) {
          symbol(ttname);

          drawrot("IP",0,28,3);
          layer(METAL);
          box(88,16,92,20);
          if(inA == 1)
               drawrot("IPD",0,14,3);
          else {
               drawrot("IP",0,14,3);
               layer(METAL);
               box(88,2,92,6);
          }
          if(inB == 1)
               drawrot("IPD",0,56,3);
          else {
               drawrot("IP",0,56,3);
               layer(METAL);
          }
          if(inC == 1)
               drawrot("IPD",0,42,3);
          else {
               drawrot("IP",0,42,3);
               layer(METAL);
               box(88,30,92,34);
          }

          drawrot("OPBI",0,84,3);
          drawrot("OPBI",0,112,3);
          x=92 ; y=0;

          if(n == 1)
               draw("ADD",x,y);
          else {
```

any cell design environment comprising cell editors, a circuit extractor, circuit simulators, etc. using well documented methods for IC design. In designing the leaf cells their internal composition interfaces must be formalised and verified. This results in a formal expression of the rules of composition. Again this is interfaced to the compiler as a piece of code that can be compiled and linked with the rest of the compiler programs. The multiplier has been illustrated in Chapter 3 (q.v.).

The parameters associated with an the ADD primitive are the latency, and the predelay on each of the inputs Thus the selection and placement of appropriate input connections or predelays must be made according to these parameter values. Then the appropriate choice of adder followed, if necessary, by the correct number of repeated blocks of delay has to be made, again according to the appropriate parameter value. Such informal rules of composition are formalised into code as shown in Table III.5.

systems can currently be developed and tested as multiple chip systems, in older technologies.

Each leaf cell is represented as a separate numbered symbol. A complete list of the numbered symbol definitions together with their bounding box sizes is stored in file for use by the compiler. The portion of this relating to the adder primitive is shown in Table III.4.

```
!external symbol spec definitions

external symbol spec("IP",1,0,0,14,92)
external symbol spec("IPD",2,0,0,14,92)
external symbol spec("OPBI",3,0,0,28,92)
external symbol spec("OPBN",4,0,0,28,92)

external symbol spec("ADD",31,-92,0,144,124)
external symbol spec("SUB",32,-92,0,144,124)
external symbol spec("ADDDEL",33,-92,0,144,124)
external symbol spec("SUBDEL",34,-92,0,136,138)
external symbol spec("ASDELA",35,0,0,46,138)
external symbol spec("ASDELB",36,0,0,46,138)
external symbol spec("ASDELC",37,0,0,46,138)
external symbol spec("ASDELD",38,0,0,38,138)
external symbol spec("ASDELE",39,0,0,14,138)
external symbol spec("ASDELF",40,0,0,32,138)
external symbol spec("ASDELG",41,0,0,56,138)
external symbol spec("ASDELH",42,0,0,36,127)
external symbol spec("ASDELI",43,0,0,36,11)
external symbol spec("ASDELJ",44,0,0,36,11)
external symbol spec("ASDELK",45,0,0,2,4)
external symbol spec("ASDELL",46,0,0,2,11)
external symbol spec("ASDELM",47,0,0,19,11)
external symbol spec("ASDELN",48,0,0,19,11)
```

Table III.4

Methods for the design of leaf cells are not discussed here as they are adequately dealt with elsewhere. This may proceed within

should be defined in terms of a technology independent intermediate form for specifying circuits. Secondly, programs for generating technology specific layouts from these descriptions should then be available. At the time of implementing FIRST, such intermediate forms and generators were not available, so it was necessary to define the cells specifically at the mask level, and therefore at a technology specific level. Current research shows promise of progress in the direction of technology independent specifications for the future. The advantage that such organisation confers is that the task of translating the primitive library is reduced by orders of magnitude. The cost is that the resulting layouts may not be as area efficient as those of a custom designed library.

The design of the leaf cells is done in the context of a modular floor plan capable of accepting their appropriate composition according to the relevant parameter values. The final form of the circuit blocks is the layout definition in the form of CIF code specifying the detailed mask geometry. The library discussed in this chapter has been constructed according to enhanced Mead - Conway design rules, for 4, 5 or 6 micron polysilicon gate NMOS process technology. This choice of technology was dictated by the facilities available for the development and proving of FIRST. For state of the art VLSI systems, FIRST will require 2 micron or sub 2 micron two level metal CMOS technology or 1 micron or sub 1 micron NMOS technology. However, since bit serial architectures and the FIRST methodology simplify system partitioning, future one chip

sigout, and param. The fourth group of lines checks for synchronisation and warns of LSB mismatching of input signals, flagging node and time for ease of diagnostic purposes. The fifth group of lines fetches the values on each of the inputs by accessing the relevant parts of the record arrays and then computes the function of the primitive, using these values. In this context there is also a mechanism for retrieving an arbitrary depth of previous state information, if needed, and also for updating the state information. This is not needed to model the adder function. If required, a further group of lines of code can be inserted to check the values for format etc, and diagnostic warnings can be issued. This is shown in the sixth group of lines. This can be a useful feature during system design and although the hardware will not perform this check, it gives a useful mechanism for flagging detectable conditions which might give rise to unexpected system behaviour. Usually such conditions are less easily traced during simulation when there are no such explicit diagnostic warnings. The seventh group of lines enter the primitive output values onto appropriate data records and event queue by means of the function enter_event. Finally the else condition of the initial if is evaluated to check primitive input activity when the control is not active.

The next stage of primitive design is to create the logic and circuit blocks or leaf cells required to make up the primitive. Ideally this should be done in two stages. Firstly the circuit

```
            else
                total = aval - (bval + cin);
            if((total > mantissa) && (co <= NC)) {
                sprintf(mess,"OVERFLOW IN ADD/SUB #%d @ %s",n,
                    bit_time(time));
                warning(mess);
            }
            if((total < sign_extend) && (co <= NC)) {
                sprintf(mess,"UNDERFLOW IN ADD/SUB #%d @ %s",n,
                    bit_time(time));
                warning(mess);
            }

            enter_event(s,time+delay,total);
            enter_event(co,time+(wordlength<<1),carry);

        }
        else {
            if((pda + pdb + pdc) == 0)
                timing_warning(n);
        }
}
```

Table III.3

The lines of code are grouped with consecutive groups separated by a single blank line. The general structure is as follows. The first group of lines after the declarations invoke the standard simulator functions ctlin, and param to find the primitive's lsb control node number and the values of the predelay parameters. These functions, together with others mentioned below, are the means whereby the primitive function code communicates with the rest of the simulator code. The values returned are allocated to local variables as access to them in this way is faster than repeated function calls. Next a test is made to determine whether there is an event on the primitive control node. The third group of lines fetches the remaining node numbers and parameter values associated with the primitive by means of the functions sigin,

```
add(n)
{
int a, b, s, delay, total, carry, cin;
int c, co, pda, pdb, pdc, clsb, aval, bval;
char *bit_time(),mess[64],mess1[256];

     clsb = ctlin(n,1);
     pda = 2 * param(n,2);
     pdb = 2 * param(n,3);
     pdc = 2 * param(n,4);

     if(nodelist[clsb + j_max_ctl].time == time) {

          a = sigin(n,1);
          b = sigin(n,2);
          c = sigin(n,3);
          s = sigout(n,1);
          co = sigout(n,2);
          delay = param(n,1) << 1;

          if(((a > NC && (nodelist[a + j_max_ctl].time + pda) != time) ||
             (b > NC && (nodelist[b + j_max_ctl].time + pdb) != time) ||
             (c > NC && (nodelist[c + j_max_ctl].time + pdc) != time)) &&
             time >= op_start) {
               timing_warning(n);
               sprintf(mess1,"----> Node #%d VALID @ %s",a,
                    bit_time(nodelist[a + j_max_ctl].time + pda));
               sprintf(mess,",Node #%d VALID @ %s",b,
                    bit_time(nodelist[b + j_max_ctl].time + pdb));
               strcat(mess1,mess);
               sprintf(mess,",Node #%d VALID @ %s0,c,
                    bit_time(nodelist[c + j_max_ctl].time + pdc));
               strcat(mess1,mess);
               trap(mess1);
          }

          aval = nodelist[a + j_max_ctl].value;
          bval = nodelist[b + j_max_ctl].value;
          cin =  nodelist[c + j_max_ctl].value & 1;
          total = aval + bval + cin;
          if((total & carry_bit) == 0)
               carry = 0;
          else
               carry = 1;

          if((aval & sign_bit) != 0)
               aval |= sign_extend;
          if((bval & sign_bit) != 0)
               bval |= sign_extend;
          if(sign == PLUS)
               total = aval + bval + cin;
```

of this function. The testing of the code can be done by installing it into the FIRST compiler (by compilation and linking) and then by making up a system specification consisting of a chip, or chips with instances of the primitive. These are then stimulated by appropriate test patterns to check their response.

An example of the form such code takes, and the way it interfaces into the overall structure of the compiler is shown in Table III.3 which gives the code of the functions defining the ADD primitive.

addition of new primitive syntax to the compiler a trivial matter. The form required for this is illustrated in Table III.2 which shows the definitions for ADD. Each consists of a single line per primitive containing, in order, the primitive name (alphanumeric), the primitive type code, the number of parameters, signal inputs, signal outputs, control inputs and control outputs. Each of these fields of information in the line is separated by a space, or spaces. This allows the compiler to recognise the primitive name, where it occurs and to check that it is being used correctly, and with the right number of parameters, inputs, outputs, etc.

```
"ADD" 2 4 3 2 1 0
```

Table III.2

In designing a primitive, the first task is to define the function required. This is achieved by formulating the word level behaviour of the primitive. This definition may have to be iteratively changed as design proceeds, since the final form will reflect on the one hand the arithmetic function required by the system designer and on the other hand the tradeoffs which the circuit designer can make to achieve hardware efficiency and minimal circuitry. In any event the final interface to the compiler is a piece of program code in the form of a function (or routine) which can be compiled and linked to the main compiler programs. Verification requires that the composition and layout information combine, when processed, to yield the behaviour captured in the code

1.8. Primitive Design and Entry

Primitive design consists firstly of designing a set of circuit elements from which a primitive can can be assembled. These are termed leaf cells. This must be done in the context of a modular floor plan which allows appropriate composition of leaf cells according to parameter value(s). The second design task is the formulation of the procedural rules of composition. A necessary part of design at this level is the verification of the internal composition interfaces. The third design task is to extract the generalised primitive behaviour from the circuits which result from composition of leaf cells. As a result of these design tasks the formal interfaces to the FIRST silicon compiler can be defined. These are:

primitive syntax
primitive functional behaviour
primitive composition
primitive leaf cell definition

As an example of the structure of each of these consider the definition of the ADD primitive, giving the full definition of each interface.

Primitive syntax is defined for the compiler in a data file reserved for this purpose. This file is read at the start of compilation and sets up the definition and attributes of the predefined primitives. This information is held as a file to make

1.7.3. Clock generator

Description:

During layout generation there is a run time option allowing selection of one of the following configurations:

either two clock pads and corresponding clock distribution for input of externally generated two phase clocks,
or alternatively one clock input with on chip two phase clock generation and distribution.

Circuit Design

Not yet available.

1.7.4. Substrate bias generator

Description:

During layout generation there is a run time option allowing selection of one of the following configurations:

no substrate bias generation (for use with multiproject EMF standard frames)
on chip substrate bias generation.

Circuit Design

Not yet available.

Restrictions:

1. Padout inputs must not be connected to GND or VSS, if the simulator is to function correctly.

Simulator Checks and Warnings

Node connectedness checks are made.

Circuit Design

The circuit design of padout is that of a modified Xerox-Parc output pad, the modification involving only the addition of a clocked buffer and the necessary changes to metalisation to allow rotated placement.

1.7.2. Padout

Description:

Padout is the mechanism for getting signals and controls off a chip and provides the correct drive and buffering capability for doing this. Communication between chips is pipelined, each chip boundary involving one half phase of the bit clock for data transfer.

Syntax:

```
padout (cc1 -> c1)
padout ss1 -> s1
padout (c1,c2,... -> cc1,cc2,...) s1,s2,... -> ss1,ss2,...
```

Note

The first form is for a single control output. cc1 is the on-chip, internal input node name and c1 the off-chip, external output node name. Similarly the second form is for a single signal output, with ss1 and s1 corresponding to cc1 and c1 respectively. The third form is the combined condensed form for multiple control and signal outputs.

Function:

output = corresponding input

latency = 1/2 bit

Restrictions:

1. For the simulator to function correctly all inputs should be connected to an appropriate signal or control source. This may be either an output pad output node or an input file, corresponding to externally supplied signal generation.

Simulator Checks and Warnings

Node connectedness checks are made.

Circuit Design

The circuit design of padin is that of a modified Xerox-Parc input pad the modification involving only the addition of a clocked driver and the necessary changes to metalisation to allow rotated placement.

1.7.1. Padin

Description:

Padin is the mechanism for getting signals and controls onto a chip and provides the correct drive and buffering capability for doing this. Communication between chips is pipelined, each chip boundary involving one half phase of the bit clock for data transfer.

Syntax:

```
padin (c1 -> cc1)
padin s1 -> ss1
padin (c1,c2,... -> cc1,cc2,...) s1,s2,... -> ss1,ss2,...
```

Note

The first form is for a single control input. c1 is the off-chip, external input node name and cc1 the on chip, internal output node name. Similarly the second form is for a single signal input, with s1 and ss1 corresponding to c1 and cc1 respectively. The third form is the combined condensed form for multiple control and signal inputs.

Function:

output = corresponding input

latency = 1/2 bit

1.7. MISCELLANEOUS PRIMITIVES

There are two types of service primitive the I/O pad primitives and the clock and subsrate bias generators. The former are visible at the level of the FIRST HDL the latter are not, being selectable as options during run time of the physical design subsystem.

param	min	max	constraint	meaning
1	1	32	integer	system word-length
2	1	n-1	integer	limit position
3	0	1	integer	predelay on input

Function:

output = input if -k < input < k

output = k if input >= k

output = -k if input <= -k

latency = swl + 1

Restrictions:

1. For correct function parameter 1 must be the same as the system word-length.

2. ctrl must be c1 control.

Simulator Checks and Warnings

Circuit Design

The circuit implementation of flimit comprises of the following components: a storage shift register, maximum and minimum value generation circuits, over-flow detection circuitry and the final output select and control circuits. Modularity ensures that these components can be assembled according to the user chosen values of parameters 1 and 2 to carry out the defined function of flimit.

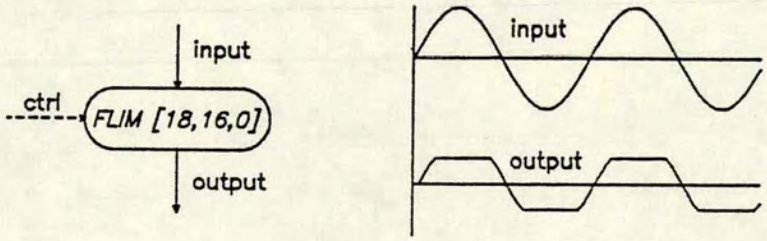
1.6.3. Flimit

Description:

Signal maximum and minimum values are defined by the user chosen values of parameter 1. If the input signal exceeds either then flimit hard limits the output signal to the appropriate limit value. Otherwise the output value is the same as the input value. Note that the maximum and minimum values are symmetrically placed, about zero, not as in strict two's complement representation.

Flimit is designed to deal with problems of overflow and dynamic range limitation.

Flow Diagram & Simulation Pattern:



Syntax:

Flimit[swl,lim,del1] (ctrl) input -› output

Parameters:

Let n=swl, let m=lim and let $k=2^m-1$

Function:

LSWout = input

MSWout = 0 if MSB of previous input = 0

MSWout = 1 if MSB of previous input = 1

LSWout latency = 1 bit

MSWout latency = (swl + 1) bits

Restrictions:

1. ctrl must be c1 control.

Simulator Checks and Warnings

Checks are made for input synchronisation timing.

Circuit Design

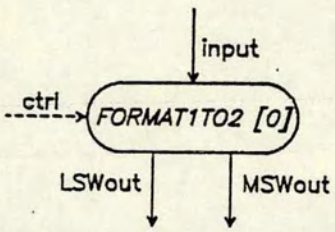
Not yet available.

1.6.2. Format1to2

Description:

Format1to2 accepts one input and outputs this input on LSWout with a one bit latency. The MSB of each input is latched and output for a full c1 cycle and output on MSWout. Thus format1to2 takes in a single precision format signal value and outputs the corresponding double precision format value, with a full word of MSB repetitions. It interprets the input as a two's complement number.

Flow Diagram & Simulation Pattern:



Syntax:

Format1to2[del] (ctrl) input -> LSWout, MSWout

Parameters:

param	min	max	constraint	meaning
1	0	1	integer	input predelay

Simulator Checks and Warnings

Checks are made for illegal parameter values (not system word-length, or out of range), and for input synchronisation timing.

Circuit Design

Not yet available.

$$X = x_0.2^0 + x_1.2^1 + \ldots + x_{n-2}.2^{n-2} + x_{n-1}.2^{n-1}$$

Note the unsigned integer interpretation of X, the LSWin.

Let

$$Y = y_0.2^0 + y_1.2^1 + \ldots + y_{n-2}.2^{n-2} - y_{n-1}.2^{n-1}$$

and let

$$f = f_0.2^0 + f_1.2^1 + \ldots + f_{2n-2}.2^{2n-2} - f_{2n-1}.2^{2n-1}$$

Define

$$g = g_0.2^0 + g_1.2^1 + \ldots + g_{2n-2}.2^{2n-2} - g_{2n-1}.2^{2n-1}$$

where

$g = f$ if $-(2^m-1) < f < (2^m-1)$
$g = -(2^m-1)$ if $f \leq -(2^m-1)$
$g = (2^m-1)$ if $f \geq (2^m-1)$

Then Z is defined as follows:

$$Z = z_0.2^0 + z_1.2^1 + \ldots + z_{n-2}.2^{n-2} - z_{n-1}.2^{n-1}$$

where $z_j = g_{j+k}$

and

latency = 1 bit

Restrictions:

1. For correct function parameter 1 must be the same as the system word-length.

2. ctrl must be c1 control.

Syntax:

Fformat2to1[swl,lim,shift,del1,del2] (ctrl) LSWin, MSWin -> output

Parameters:

Let n=swl, m=lim, k=shift

param	min	max	constraint	meaning
1	1	32	integer	system word-length
2	0	63	integer	limit pointer
3	0	63	integer	shift pointer
4	0	1	integer	LSWin predelay
5	0	1	integer	MSWin predelay

Note

Parameter 2, the limit pointer, points to the bit position of the assembled double precision word which is to be the MSB of the limited single precision output. This defines hard limiting and overflow.

Parameter 3, the shift pointer, points to the bit position of the assembled double precision word which is to be the LSB of the single precision output. This defines the power of two shift value.

A value of 0 points to LSB of LSWin

A value of n points to LSB of MSWin

Function:

For notational convenience let

Fformat2to1[n,m,k,0,0] (C) X, Y -> Z

Let

```
Fformat1to1[swl,lim,shift,del] (ctrl) LSWin -› output
Fformat2to1[swl,lim,shift,del1,del2] (ctrl) LSWin, MSWin -› output
Fformat3to1[swl,lim,shift,del1,del2,del3] (ctrl) -
              LSWin, MSWin, HSWin -› output
```

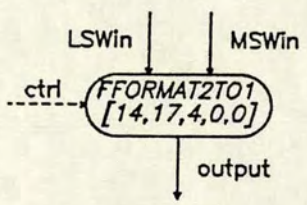
Fformat2to1 is now defined in the same way as the previous primitives to illustrate this in detail.

Fformat2to1

Description

Fformat2to1 accepts two inputs on LSWin and MSWin staggered by one word cycle. It interprets the first as magnitude only and the second as two's complement and assembles them into a double precision word. It then hard limits any overflow or underflow condition defined by the user chosen value of parameter 2. Finally it outputs single precision word whose start bit is defined by the user chosen value of parameter 3.

Flow Diagram & Simulation Pattern:



1.6.1. Fformat1to1, Fformat2to1, Fformat3to1

Description

Fformat is a fixed format primitive which accepts and assembles a multiple precision input and then performs a windowing operation to extract a single precision output word from this input. The windowing consists of doing overflow detect (as determined by parameter 2, limit) clamping if overflow has occurred, and then extracting the appropriate n bits of the multiple precision word (as determined by parameter 3, shift). In the syntax swl=n is the system word length; lim is the pointer to the bit position of the multiple precision word which is to be the MSB of the limited single precision word; and shift is the pointer to the bit position of the multiple precision word which is to be the LSB of the single precision output. The output selects n consecutive bits from the bit defined by parameter 3. A pointer value of zero points to LSB of LSWin; a pointer value of n points to LSB of MSWin; a pointer value of 2n points to LSB of HSWin, etc.

The latency of the primitive is

latency = (p*n + 1) bits

where n=swl and p=1,2,3 for Fformat1to1, Fformat2to1, Fformat3to1 respectively.

Syntax

1.6. FORMAT TRANSFORMATION PRIMITIVES

Fixed-point, bit-serial arithmetic imposes constraints on the data input formats to hardware. In particular, for example, the inputs to adders must have the same scaling; inputs to accumulation may only require single precision, but the accumulation may need to be double or triple precision; multiple precision output may have to be scaled, limited, or reduced to single precision, etc. The primitives described in this section are designed to meet these requirements. They comprise the following. There is a range of Fformat primitives (standing for Fixed format), which reduce multiple precision input to single precision, with scaling and overflow detect and clamping. There is a range of format primitives, which take single precision input and give multiple precision output (using MSB sign repetition). Finally there is a limiter, flimit, for detecting overflow and clamping. Additions to this range of primitives would include Pformat primitives (programmable format) and another version of the limiter.

signals, can be shown to constitute guinea delay.

Restrictions:

1. A latency of one guinea is a latency of one word plus one bit.

2. The maximum value of parameter 1 is determined only by floor plan considerations, typically values of up to 32 might be used.

3. LSBctrl input must be c1 control; ctrlin should be c2 or a higher level of control. It is unnecessary and inefficient to propagate c1 through worddelay.

4. Because of the internal architecture of cworddelay the input pulses must be of the form produced by the control generator (see description).

5. Do not connect the input of cbitdelay to VDD or GND.

Simulator Checks and Warnings

No checks are made.

Circuit Design

As has been mentioned the basic circuit function block senses the MSB from the previous word and latches this, outputting it one bit time later for the duration of one word. This can be implemented by sixteen transistors per guinea delay compared to 6(n+1) transistors, where n = swl. Where w is the value of the first, w such blocks are cascaded.

latency = m(n + 1) bits

The latency of cworddelay is determined by the user chosen value of parameter 1.

Because it is designed for use with the special periodic waveforms generated by the control generator it is not necessary for it to store all the bits of each control word. By sensing the MSB of the previous control word and latching it for one word time the same function can be implemented with much reduced hardware. More formally if we have, n = swl and

cword[1,0](C,X -> Y)

and if

$$X = x_0.2^0 + x_1.2^1 + \ldots + x_{n-2}.2^{n-2} - x_{n-1}.2^{n-1}$$

and Y is given by

$$Y = y_0.2^0 + y_1.2^1 + \ldots + y_{n-2}.2^{n-2} - y_{n-1}.2^{n-1}$$

where

$$C = 2^n - 1$$

then :

$Y(t) = 0$ if $x_{n-1}(t-1) = 0$
$Y(t) = 2^n - 1$ if $x_{n-1}(t-1) = 1$

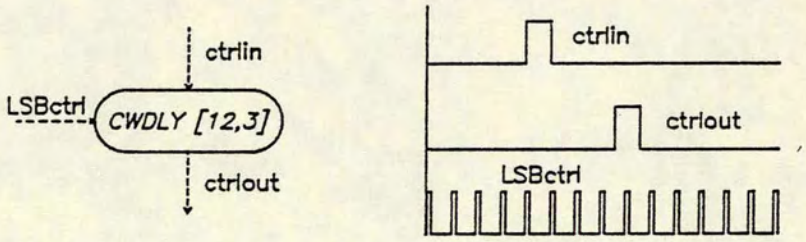
In other words the output is zero if the MSB of the previous input word was zero and the output is all ones if the MSB of the previous input word was one. If the first parameter is w then the output word depends on the MSB of the w-th previous word. This bit level definition of function, together with the definition of the control

1.5.3. Cworddelay

Description

Cworddelay is a special bit serial FIFO storage primitive. It gives storage with a latency in "guineas" (multiples of one word plus one bit).

Flow Diagram & Simulation Pattern:



Syntax:

cworddelay [w,del] (LSBctrl, ctrlin -> ctrlout)

Parameters:

param	min	max	constraint	meaning
1	1	*	integer	latency in "guineas"
2	0	1	integer	predelay on input

Function:

ctrlout = ctrlin

(see restrictions and description).

values of the parameter n. The event control generation is achieved by use of a some detect circuitry, a latch and reset circuitry associated with the appropriate level of cycle.

Restrictions:

1. All systems must have one and only one CONTROLGENERATOR.

2. Only INH, the inhibit output and c1, the c1 level control pulse are mandatory.

3. A cycle statement at any level must be preceded by cycle statements at all previous levels.

4. Controlgenerator should not be called in an operator.

5. Note that the simulator output on INH does not correspond to the actual hardware signal.

Simulator Checks and Warnings

No checks are made.

Circuit Design

The circuit design of controlgenerator is based on a divide-by-n counter (where n is the count period) which is used to implement each cycle. The counter is implemented as a synchronous binary counter, made up of a series of set-reset toggle flip flops. These are chained with a combination of parallel and series carry chains, a maximum count detect and counter reset circuit. The choice of this counter architecture over alternatives was dictated by the fact that it gave the best trade-off between low area, high speed and modularity of layout composition for the required range of

asynchronous requests.

The problem of interfacing FIRST bit-serial systems to the real-time sample and hold hardware and other systems can be approached in two ways. Firstly a single fast system clock can be divided down and used as required throughout the system. This is simple and reliable but may be inflexible. Where such a solution is either not possible or inappropriate it may be necessary to provide a synchronisation facility between the bit serial system clock and that of other systems. This is done by allowing the bit-serial system to operate in "burst" mode by computing a full sample cycle and then, on completion inhibiting its clock until the start of the next sample. The INH output of the control generator provides a "cycle complete" signal for use in such clock start-stop circuits.

To illustrate the syntax of controlgenerator, consider the following instance.

```
controlgenerator(evr -> INH, c1, c2, ev, C3)
     cycle[5]
     cycle[4]
     cycle[2]
     event
endcontrolgenerator
```

The simulation pattern shows a timing diagram of the outputs of this control generator and the effect of an asynchronous request on the input evr.

dividers. The lowest level of hardware control is the two phase bit-clock which is invisible to the system designer. The next level of control is c1, the LSB marker pulse, which is high for the duration of one bit time per word, in coincidence with LSB, and low for the rest of the time. The next level of control is c2, which is normally used to control multiplexing operations. The cycle pattern of c2 is high for the first word of the cycle and low for the remainder. Thus c2 marks and counts units of c1. Similar, higher order control cycles can be generated above c2. Non-cyclic control pulses which are high for an initial interval corresponding to any of the levels of cyclic control and then low indefinitely thereafter are required for reset, initialisation etc. Such pulses need to be available as a result of an external, asynchronous request, ER1, ER2, ... etc. This type of control is defined as an event.

In the syntax for specifying CONTROLGENERATOR, the body of the definition consists of an arbitrary number of cycle statements, each of which can be optionally followed by an event statement. Each cycle statement has a single user chosen parameter which defines the period of that control cycle. The period counts units of the previous cycle length. Each cycle has one output, each event has an event request input and an event output. The first level of control is c1. This defines the system wordlength and is high during the LSB and low until the next LSB. All outputs from the controlgenerator are synchronous. The event inputs can be

Parameters:

param	min	max	constraint	meaning
period	2	256	integer	cycle length

where period = per1, per2,...

Note:

Controlgenerator is a primitive with a variable number of inputs and outputs. Event request inputs ER1, ER2,... their associated outputs E1, E2,... and their associated event statement are optional. (To each ER1, ER2, ER3,... there corresponds respectively E1, E2, E3,... and an event statement at the appropriate level. To each c1, c2,... there corresponds a cycle(per1), cycle(per2),... statement.)

Function:

$c1 = 1$ for all values of t

$c2 = 2^n$ if T = 0 mod (per2)

$= 0$ otherwise

$c3 = 2^n$ if T = r mod (per1.per2) where r = 0, 1,..., (per2 - 1)

$= 0$ otherwise

where time reference starts at T=0 when the first word of each control is produced at the controlgenerator output.

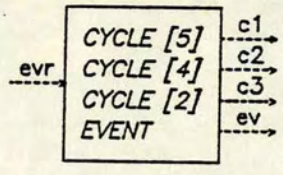
Because of the cyclic nature of the control pulses the control generator can be implemented as a bank of counters, or frequency

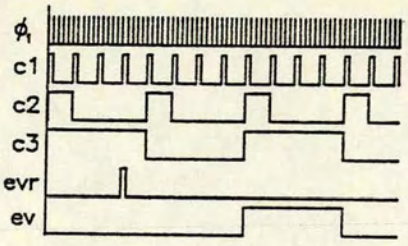
1.5.2. Controlgenerator

Description:

Special purpose hardware is needed to generate the control pulses required to keep bits, words, multiplexing and initialisation operations etc, in synchronisation. This is the function of the CONTROLGENERATOR primitive, which produces c1, c2, ... and event levels of control

Flow Diagram & Simulation Pattern:





Syntax:

```
controlgenerator(ER1,ER2,... -> INH,c1,E1,c2,E2,...)
     cycle[per1]
     event
     cycle[per2]
     event
      :
      :

endcontrolgenerator
```

Function:

output = input

latency = user chosen value of parameter 1

Restrictions:

1. Cbitdelay does not alter the value of the input in transferring it to the output, it acts as a storage FIFO.

2. The maximum value of parameter 1 is determined only by floor plan considerations, typically a value of less then or equal to 24 is appropriate if the largest primitive is a 16 coefficient bit multiplier.

Simulator Checks and Warnings

No checks are made.

Circuit Design

Cbitdelay is implemented as a shift register using a six transistor bit leaf cell. The input predelay is used as first cell in all instances of bit delay with latency greater than 1. The output buffer forms the last half cycle of delay.

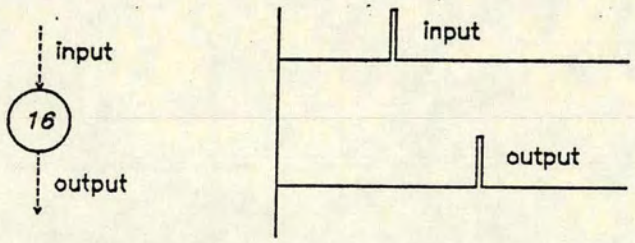
1.5. CONTROL PRIMITIVES

1.5.1. Cbitdelay

Description:

Cbitdelay is a storage primitive for control. It is a first in first out bit serial memory. The main system use of cbitdelay is for implementing compensating delay to synchronise control streams. In particular it is used to make delay lines from which appropriately delayed versions of control can be tapped.

Flow Diagram & Simulation Pattern:



Syntax:

cbitdelay [latency] (input -> output)

Parameters:

param	min	max	constraint	meaning
1	1	32	integer	latency of output

Simulator Checks and Warnings

Checks are made for coefficient (out of range) and for input synchronisation timing.

Circuit Design

Worddelay is based on the three transistor RAM cell. The memory block architecture consists of a serial to parallel input register, a word organised block of RAM and a parallel to serial output register. The memory is organised as a FIFO with data items traversing the memory locations. This obviates address generation and the control associated with addressing. The input and output registers are clocked by c0, moving n bits serially each word time. The internal RAM is clocked by c1, moving blocks of words in RAM each bit time in such a way as to move all once per word time.

Parameters:

param	min	max	constraint	meaning
1	8	*	integer	number of words storage
2	1	swl	integer	number of significant bits
3	0	1	integer	predelay on input

Function:

Let swl=n.

If bit m of input = 0 then:

output = input AND (2^m-1)

If bit m of input = 1 then:

output = input AND $((2^n-1)-(2^m-1))$

latency = (m*n + 1) bits

The latency of worddelay is determined by the user chosen value of parameter 1.

Restrictions:

1. The maximum values of parameter 1 is determined only by floor plan considerations, typically values of between 12 and 64 might be used.

2. The value of parameter 2 must be less than or equal to the system wordlength.

3. Do not connect the input of bitdelay to VDD or GND.

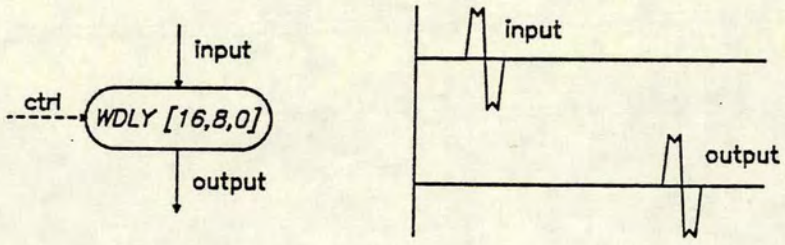
4. Ctrl must be LSB control c1, for correct function of the word-delay primitive.

1.4.2. Worddelay

Description:

Worddelay is a word oriented FIFO storage primitive. It is specially organised for efficient storage of signals which utilise less than the full dynamic range of the system word-length, for example coefficients which do not require full precision quantisation. Word delay can be configured to store only the m least significant bits of each input word. It assumes that bit m of the input word is the sign bit and that the remaining bits are identical sign bit repetitions. It uses bit m on output to replicate these sign bit repetitions.

Flow Diagram & Simulation Pattern:



Syntax:

worddelay [w,m,del] (ctrl) input -> output

Function:

output = input

latency = user chosen value of parameter 1

Restrictions:

1. The output and input have the same significance.

2. Bitdelay does not alter the value of the input in transferring it to the output. It acts as a storage FIFO.

3. The maximum value of parameter 1 is determined only by floor plan considerations, typically a value of less then or equal to 24 is appropriate if the largest primitive is a 16 coefficient bit multiplier.

4. Do not connect the input of bitdelay to VDD or GND.

Simulator Checks and Warnings

No checks are made.

Circuit Design

Bitdelay is implemented as a shift register with a six transistor per bit leaf cell. The input predelay is used as first cell in all instances of bit delay with latency greater than 1. The output buffer forms the last half cycle of delay.

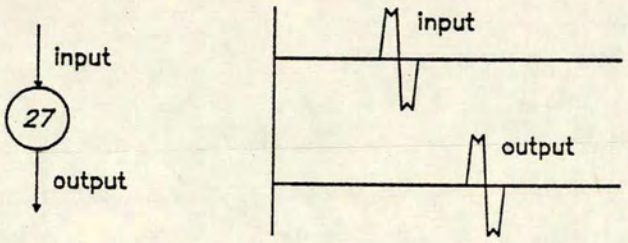
1.4. STORAGE PRIMITIVES

1.4.1. Bitdelay

Description:

Bitdelay is a storage primitive. It is a first in first out (FIFO) bit serial memory, implemented as a shift register. The main system use of bitdelay is for implementing compensating delay to synchronise signal data streams (cp. cbitdelay).

Flow Diagram & Simulation Pattern:



Syntax:

bitdelay [latency] input -> output

Parameters:

param	min	max	constraint	meaning
1	1	32	integer	latency of output

The LSB of the difference will have this significance also.

2. If the minuend and the subtrahend have one guard bit (i.e., the two MSBs are identical) then no overflow can occur.

3. Two word subtraction can cause arithmetic growth of one bit. Overflow in the previous difference is flagged by the single bit value of carryout at LSB time, with value 0 - no overflow and value 1 - overflow.

4. For single precision subtraction the borrrow input is connected to ground.

5. For correct function the ctrlin input must be LSB or c1 control.

Simulator Checks and Warnings

Checks are made for overflow and underflow and for input synchronisation timing.

Circuit Design

As for adder primitive (see section 4.5.2 circuit design).

The latency of the difference can be user chosen by means of the first parameter.

F is the word of bitwise subtraction bits generated from the subtraction, and has a fixed latency of only one bit.

More formally, if

$$A = a_0.2^0 + a_1.2^1 + \ldots + a_{n-2}.2^{n-2} - a_{n-1}.2^{n-1}$$
$$B = b_0.2^0 + b_1.2^1 + \ldots + b_{n-2}.2^{n-2} - b_{n-1}.2^{n-1}$$
$$C = c_0.2^0 + c_1.2^1 + \ldots + c_{n-2}.2^{n-2} - c_{n-1}.2^{n-1}$$

then F is given by

$$F = f_0.2^0 + f_1.2^1 + \ldots + f_{n-2}.2^{n-2} - f_{n-1}.2^{n-1}$$

where :

$$f_1 = c_0 \text{ AND } (a_0 \text{ XNOR } b_0) \text{ OR } ((\text{ NOT } a_0) \text{ AND } b_0)$$

$$f_j = f_{j-1} \text{ AND } (a_{j-1} \text{ XNOR } b_{j-1}) \text{ OR } ((\text{ NOT } a_{j-1}) \text{ AND } b_{j-1})$$

where $1<=j<=n-1$ and

$$f_0 = f_{n-1} \text{ AND } (a_{n-1} \text{ XNOR } b_{n-1}) \text{ OR } ((\text{ NOT } a_{n-1}) \text{ AND } b_{n-1})$$

Note that, in this final expression for f_0 the values a_{n-1} and b_{n-1} refer to the previous input words and f_{n-1} the corresponding previous word carry bit. Thus f_0 is an overflow flag for the previous subtraction.

Restrictions:

1. For subtraction to function correctly the LSB of the minuend, the subtrahend and of the borrowin must have the same significance.

param	min	max	constraint	meaning
1	1	32	integer	latency of difference
2	0	1	integer	predelay on minuend
3	0	1	integer	predelay on subtrahend
4	0	1	integer	predelay on borrowin

Function:

difference = minuend - subtrahend - (borrowin & 1)

latency = user chosen value of parameter 1

borrowout: see below

latency = 1 bit

For notational convenience, suppose we have:

subtract[m,0,0,0](cLSB) A,B,C -> D,F

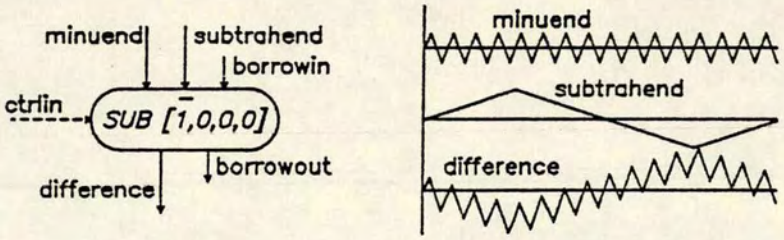
and let swl=n. Then the minuend, A, and subtrahend, B, are assumed to have the same interpretation i.e., LSBs have the same significance. D is the difference of A and B with only the LSB of C subtracted. In normal use C will be replaced by GND so that this LSB is zero. Subtracters can, however, be cascaded to form multiple precision subtraction, in the same way as for addition. This is achieved by connecting the borrowout from the previous subtracter to the borrowin of the next (cp. description of the add primitive). In this case the C input is used to propagate any borrow generated from the subtraction of less significant bytes into the subtraction of the more significant bytes. The first subtracter has its C input replaced by GND.

1.3.10. Subtract

Description

Subtract forms the bit-serial difference of the first two inputs. The third input is an external borrow, which is strobed at LSB time to initialise the subtracter. This input is normally connected to GND, but can be used to construct multiple precision subtracters (see function). The second output, the word of borrow bits, is provided for the same purpose. The latency of the difference output can have any user chosen value in the allowed range.

Flow Diagram & Simulation Pattern:



Syntax:

subtract [latency,del1,del2,del3] (ctrlin) -

minuend,subtrahend,borrowin -> difference,borrowout

Parameters:

Function:

max = max(in1,in2)
min = min(in1,in2)

latency = (n + 3) bits

Restrictions:

1. The value of parameter 1 must be the same as the system word-length for correct function.

2. For correct function the inputs must both be positive, MSB=0. An alternative version of order could be designed to cope with negative and positive inputs but this requires greater hardware area (see section on circuit design).

3. For correct operation ctrl must be c1.

Simulator Checks and Warnings

Checks are made for illegal values of parameter 1, input out of range (underflow) and for input synchronisation timing.

Circuit Design

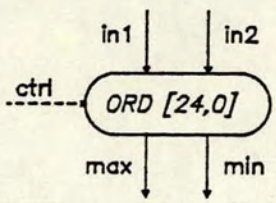
The circuit design of order requires two storage shift registers, output select circuitry and a detect and control latch circuit. Detection is achieved by a small finite state machine or serial decoder which determines which input is the larger. This is implemented in random logic using some fourteen transistors.

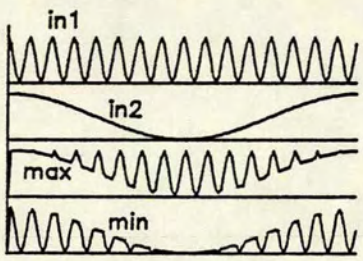
1.3.9. Order

Description:

The order primitive takes two input signals which are assumed to be positive and orders them so that the larger is output on max and the smaller is output on min. The original cell designs were done for a specific application not requiring the more general form capable of ordering both positive and negative inputs.

Flow Diagram & Simulation Pattern:





Syntax:

order[swl,del](ctrl) in1,in2 -› max, min

Parameters:

Let swl=n.

param	min	max	constraint	meaning
1	4	32	integer	system word-length
2	0	1	integer	predelay on input

repetitions.

4. For correct function ctrlin must be the correctly synchronised LSB pulse or c1 control.

Simulator Checks and Warnings

Checks are made for invalid parameter values, incorrect data format, incorrect coefficient format and for input timing synchronisation.

Circuit Design

The multiplier design is a variant of the modified Boothe serial-parallel multiplier. It differs from the Lyon multiplier in respect of recoding and output scaling. Recoding is not performed in an explicit block at the input but is distributed throughout the multiplier cells. The scaling is adapted to preserve the arithmetic output format as explained in the previous section. The generation of the rounding signal is achieved using a compact shift register in the I/O channel and is thus made invisible to the system designer. The basic leaf cells are two bit multiplier cells, composed from a recoder cell, a programmable add-subtract cell and some shift register cells. A block diagram is shown in Figure 3.9 and an instance of circuit layout in Figure 3.10 (both in Chapter 3).

The consequence of this is that if the data is interpreted as being fractional 2's complement with 2 guard bits (sign repetition) and the coefficient is interpreted as being m bit fractional 2's complement with the m-th bit as sign bit then the product will be fractional 2's complement with 2 guard bits. This corresponds to the form in which the multiplier is often most conveniently configured.

In systems where the product scaling is not as required either the data or coefficient must be scaled prior to multiplication or the product must be scaled subsequently. Alternatively the double precision product might be used in conjunction with a format primitive.

Restrictions:

1. Due to the nature of the multiplication algorithm employed there must be two bits of sign extension on the data word to prevent internal overflow, i.e., the three most significant bits of the data word must be identical.

2. The parameter coeffbits should lie in the range:
4 <= coeffbits <= swl - 2

3. Because only the first (coeffbits) bits of the coeff word are used as the multiplier coefficient, it is assumed that the MSB of these is the sign bit. The remaining more significant bits making up the full coefficient word of (swl) bits are assumed to be MSB

interpretation of data, coefficient and product. Other interpretations are now dealt with.

If it is assumed that data and product have the same scaling (arbitrary, not necessarily integer, i.e., the LSB of each has the same significance) and if it is assumed that the coefficient has integer interpretation then the truncated product is:

$$\text{product}=X*Y*2^{-(m-1)}$$

If it is assumed that the data and product have the same scaling and that the coefficient is not integer but integer scaled by a factor of 2^k (where k is integer) then:

$$\text{product}=X*Y*2^{-k-(m-1)}$$

If it is assumed that the coefficient is integer and that the product is interpreted as having the significance of the data scaled by a factor of 2^p (where p is integer) then:

$$\text{product}=X*Y*2^{p-(m-1)}$$

If it is assumed that the coefficient is interpreted as integer scaled by a factor of 2^k and the product is interpreted as having the significance of the data scaled by a factor of 2^p then:

$$\text{product}=X*Y*2^{p-k-(m-1)}$$

Rounding

Figure III.2 shows how the rounded product is formed when the multiplier type parameter has value 1.

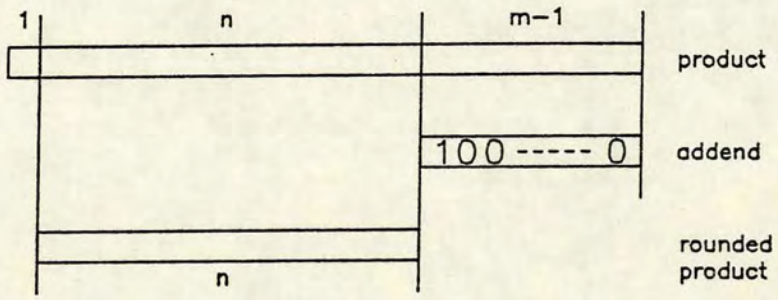


Figure III.2

Rounding is performed by adding (2^{m-2}) to the product before truncating. This is achieved by the conventional minimal hardware rounding scheme as used by Lyon. This rounding algorithm is inaccurate, having a small positive mean error. This is because the exact mid values are consistently rounded to the higher value. In system structures with feed-back this effect may be unacceptable. Where this is found to be the case different rounding hardware is needed. This requires more complex circuitry and correspondingly more chip area per multiplier. Rounding with zero mean error is selected by setting the type parameter to 2.

Interpretational Scaling

The value of the product has been expressed in terms of an integer

double precision product (cp. the dpmultiply primitive). The way in which the multiplier forms the truncated product is illustrated in Figure III.1.

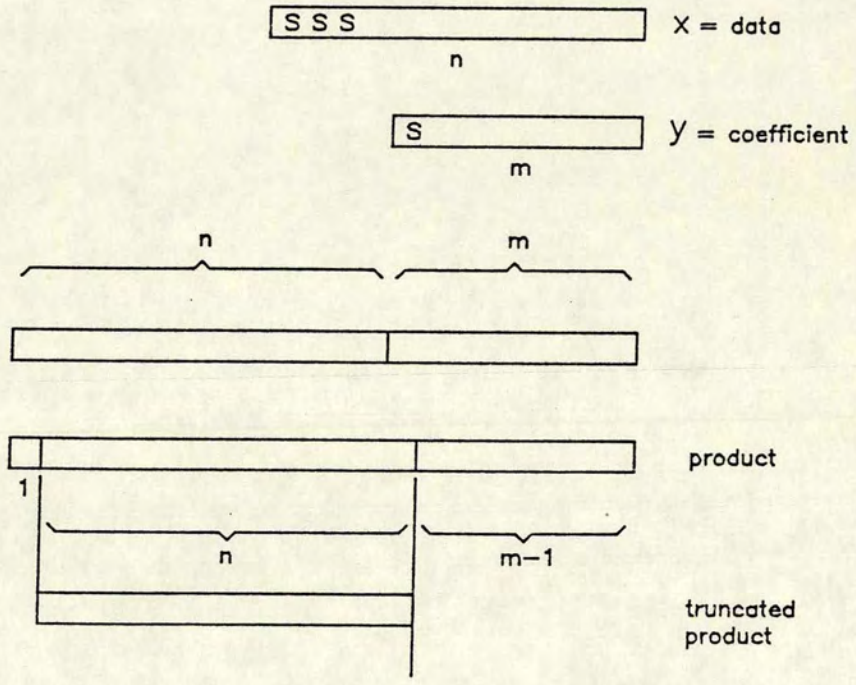


Figure III.1

param	min	max	constraint	meaning
1	0	2	integer	multiplier type 0 truncating 1 rounding 2 rounding
2	4	24	even integer	coefficient word-length
3	0	1	integer	predelay on data
4	0	1	integer	predelay on coefficient

Note: See description for details on rounding.

Function:

$$product = data * coef\ f * 2^{-(coef\ f\ bits-1)}$$

delayeddata = data

where data, coefficient & product are interpreted as integers.

latency = (1.5 * coeffbits + 2)

where this latency is that of ctrlout, product & delayeddata.

For notational convenience, suppose we have:

multiply[0,m,0,0](c1in -> c1out) X,Y -> Z,W

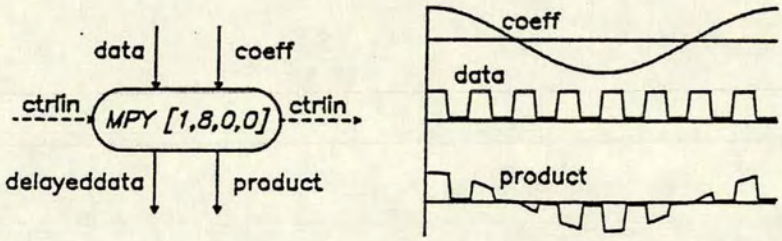
and let swl=n. Then the data X (or multiplicand) consists of n bits with the three most significant being identical (two sign bit repetitions and a sign bit). The coefficient (or multiplier) Y consists of n bits. Only the least significant m of these are used by the multiplier, the most significant n-m bits are assumed to be sign bit repetitions. The product of an n bit word with an m bit word requires (n+m-1) bits to represent it. Since the bit serial system word-length is fixed at n bits, this product can either be truncated or rounded to give n bits, or it can be split into two n bit words, a most significant and a least significant, giving a

1.3.8. Multiply

Description

Multiply forms the truncated or rounded product its the two inputs. The data (multiplicand) is assumed to have its top three bits identical. Coefficient (multiplier) quantisation is assumed to be less than or equal to that of the data. The significant bits of the coefficient are assumed to be right justified and the remaining bits are assumed to be sign bit repetitions. The input data is available, synchronously with the product, as a second output.

Flow Diagram & Simulation Pattern:



Syntax:

multiply [type,coeffbits,del1,del2] (ctrlin -› ctrlout) -
data,coeff -› product,delayeddata

Parameters:

Restrictions:

1. The output and input have the same significance.

2. Multiplex does not alter the value of the input in transferring it to the output but only selects between the two inputs according to the control word.

3. Ctrlin must be either 0 or (2^n-1) for correct functioning of multiplex. The function of multiplex for any other control input word is not a valid word but breaks down into bit-wise selection. Thus ctrlin must be level of control above c1.

Simulator Checks and Warnings

Checks are made for input synchronisation timing.

Circuit Design

The multiplex circuit is a straightforward combinational logic design, with no special features.

latency = user chosen value of parameter 1

For notational convenience, suppose we have:

multiplex [m,0,0] (C) X,Y -> Z

and let swl=n. The multiplex primitive is a data selector of one from two input streams. More formally the bit-wise function of multiplex can be defined as follows. Define the input words X, Y as:

$$X = x_0.2^0 + x_1.2^1 + \ldots + x_{n-2}.2^{n-2} - x_{n-1}.2^{n-1}$$
$$Y = y_0.2^0 + y_1.2^1 + \ldots + y_{n-2}.2^{n-2} - y_{n-1}.2^{n-1}$$

Define the control word C as:

$$C = c_0.2^0 + c_1.2^1 + \ldots + c_{n-2}.2^{n-2} - c_{n-1}.2^{n-1}$$

For each word either $c_j = 0$ for all values of j or $c_j = 1$ for all values of j.

Let the output Z be denoted by:

$$Z = z_0.2^0 + z_1.2^1 + \ldots + z_{n-2}.2^{n-2} - z_{n-1}.2^{n-1}$$

Then Z is defined as follows:

$$z_j = ((x_j)\text{AND}(\text{NOT}(c_j)))\text{OR}((y_j)\text{AND}(c_j))$$

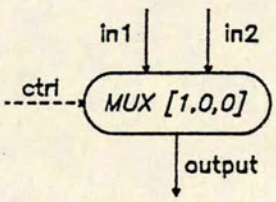
In other words if c_j is 0 then z_j is x_j or else if c_j is 1 then z_j is y_j.

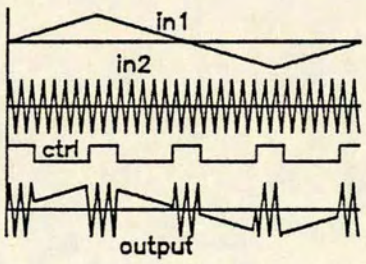
1.3.7. Multiplex

Description

Multiplex is a one of two input select switch controlled by c2 or a higher level of control. The output latency is a user chosen value.

Flow Diagram & Simulation Pattern:





Syntax:

multiplex [latency,del1,del2] (ctrlin) in1,in2 -> output

Parameters:

param	min	max	constraint	meaning
1	1	32	integer	latency of output
2	0	1	integer	predelay on in1
3	0	1	integer	predelay on in2

Function:

output = in1 if ctrlin = 0

output = in2 if ctrlin = (1 << sw1) - 1

hardware can be used. This however gives rise to primitives which have zero or negative latency. The concept of a primitive with zero or negative latency is problematic for the implementation of an event driven word level simulator. Thus for FIRST mshift has been implemented as a primitive with one bit latency and no hardware overflow detect, though the simulator gives warnings of overflow. The circuit can then be implemented as a multiplexer with one input tied to ground and the other connected to the input signal. A control delay line and distributed nor gate with inverter then make up the control function.

Function:

output = input * ($2^{p)}$

latency = 1 bits

Restrictions:

1. The parameter p should be less than (n-1), and must be less than n.

2. For correct operation ctrl must be c1.

3. For any value of p the result may be an overflow. This will occur if the (p+1) most significant bits of the input are not identical.

Simulator Checks and Warnings

Checks are made for out of range parameter values, for overflow and for input synchronisation timing.

Circuit Design

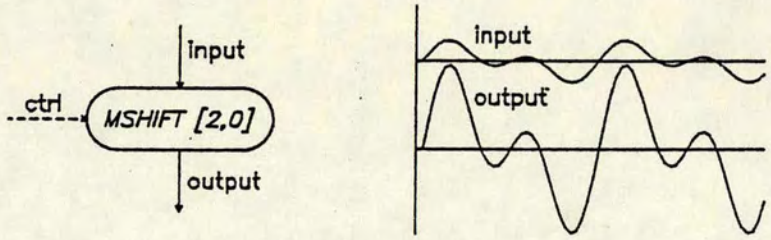
Several versions of hardware are possible, including overflow detect and a variety of latencies. Two problems arise in deciding how to implement mshift. The first problem concerns overflow and the second latency. If overflow detect is omitted then a considerable reduction in circuitry is achievable. If additionally the output bits are made available with minimum latency then minimum

1.3.6. Mshift

Description:

The mshift primitive functions as repeated arithmetic left shift, the number of repetitions being defined by the user chosen value of parameter1. In other terminology the effect is the same as multiplying the input signal by 2^p, where p is positive and integer.

Flow Diagram & Simulation Pattern:



Syntax:

mshift [p,del] (ctrl) input -> output

Parameters:

Let sw1 = n

param	min	max	constraint	meaning
1	1	n	integer	power of 2 multiply
2	0	1	integer	predelay on input

Function:

output = input / (2^p)

latency = (p + 3) bits

Restrictions:

1. The parameter p should be less than (n-1), and must be less than n.

2. For correct operation ctrl must be c1.

3. If a value of p > n-1 were to be chosen, then the division would underflow.

Simulator Checks and Warnings

Checks are made for out of range parameter values and input synchronisation timing.

Circuit Design

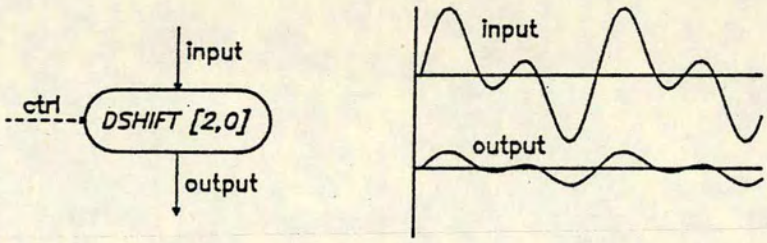
The circuit for dshift is implemented by using a one bit shift register latch in the signal path. This either passes the input signal to the output or latches a previous input. Circuitry for controlling this latch function consists of a cascadeable control delay line together with an associated distributed nor gate and inverter. Finally there is an initial input delay of one bit and an output stage.

1.3.5. Dshift

Description:

The dshift primitive functions as repeated arithmetic right shift, the number of repetitions being defined by the user chosen value of parameter1. In other terminology the effect is the same as dividing the input signal by 2^p, where p is positive and integer and then truncating to n bits.

Flow Diagram & Simulation Pattern:



Syntax:

dshift[p,del](ctrl) input -> output

Parameters:

Let swl = n

param	min	max	constraint	meaning
1	1	n-1	integer	power of 2 divide
2	0	1	integer	predelay on input

4 <= coeffbits <= swl - 2

3. Due to the fact that only the first (coeffbits) bits of the coeff word are used as the multiplier coefficient, it is assumed that the MSB of these is the sign bit. The remaining more significant bits making up the full coefficient word of (swl) bits are assumed to be MSB repetitions.

4. For correct function ctrlin must be the correctly synchronised LSB pulse or c1 control.

5. The MSWprod of a product always emerges one word later than the LSWprod.

Simulator Checks and Warnings

Checks are made for incorrect data format, incorrect coefficient format and for input synchronisation timing.

Circuit Design

The multiplier design is derived from the modified Boothe serial-parallel multiplier used for the multiply primitive. The hardware architecture of the multiply primitive forms all (n+m-1) bits of the double precision product internally, so that dpmultiply includes some additional hardware to extract these bits, together with the added control and sign extend circuitry.

consists of n bits. Only the least significant m of these are used by the multiplier, the most significant n-m bits are assumed to be sign bit repetitions. The product of an n bit word with an m bit word requires (n+m-1) bits to represent it. Since the bit serial system word-length is fixed at n bits, this product can either be truncated or rounded to give n bits (cp. multiply primitive) or it can be split into two n bit words, a most significant and a least significant, giving a double precision product. The double precision product is formed by packing the n least significant bits of the product in LSWprod and the remaining bits contiguously in MSWprod starting at LSB. The remaining bits of MSWprod are sign bit extensions.

Addend

The product output can encorporate a restricted offset added into it in the process of its formation. This is a consequence of the hardware implementation of the multiplier (cp section on circuit design).

Restrictions:

1. Due to the nature of the multiplication algorithm employed there must be two bits of sign extension on the data word to prevent internal overflow, i.e., the three most significant bits of the data word must be identical.

2. The parameter coeffbits should lie in the range:

param	min	max	constraint	meaning
1	4	32	even integer	coefficient word-length
2	0	1	integer	predelay on data
3	0	1	integer	predelay on coeff
4	0	1	integer	predelay on addend

Function:

Let n=swl and m=coeffbits then:

$$LSWprod = (data * coeff + addend) \text{ AND } (2^{n}-1)$$
$$MSWprod = (data * coeff + addend) \text{ AND } ((2^{2n}-1)-(2^{n}-1))$$

deldata = data

where data, coeff, addend & product are interpreted as integers.

Latencies ar given as follows:

ctrlout latency = (1.5 * m + 1) bits

deldata latency = (1.5 * m + 1) bits

If swl >= (1.5 * m + 1) then

LSWprod latency = 1 bit

MSWprod latency = (n + 1) bits

otherwise:

LSWprod latency = (1 + (1.5 * m + 1) - n) bits

MSWprod latency = (1.5 * m + 2) bits

For notational convenience, suppose we have:

dpmultiply[m,0,0,0](c1in -> c1out) X,Y,A -> P,Q,Z

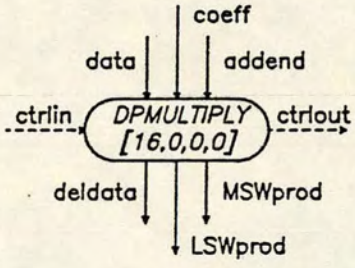
and let swl=n. Then the data X (or multiplicand) consists of n bits with the three most significant being identical (two sign bit repetitions and a sign bit). The coefficient (or multiplier) Y

1.3.4. Dpmultiply

Description:

Dpmultiply is a bit-serial multiplier which outputs a double precision product on two wires. The lower precision bits are packed, right justified, on one output and the remaining bits are packed, again right justified, on the other output, whose remaining bits are sign extensions. The two parts of the product emerge separated by a delay of one word, in keeping with the double precision format.

Flow Diagram



Syntax:

```
dpmultiply[coeffbits,del1,del2,del3](ctrlin -> ctrlout) -
        data,coeff,addend -> LSWprod,MSWprod,deldata
```

Parameters:

Circuit Design

The circuit implementation of constgen consists of a parallel load serial output m bit shift register, and control circuitry to govern the loading. The shift register has its parallel load inputs hardwired to correspond with the values of constspec and the bit m is latched to generate the (n-m) bit repetitions..

* see below

Function:

Let the base two expression for k be as follows:

$$k = k_0.2^0 + k_1.2^1 + \ldots + k_{m-2}.2^{m-2} + k_{m-1}.2^{m-1}$$

Note that m <= n and that this is unsigned integer representation, not twos complement representation. Define

$$x = x_0.2^0 + x_1.2^1 + \ldots + x_{n-2}.2^{n-2} - x_{n-1}.2^{n-1}$$

where $x_j = k_j$ for $j < m$ and $x_j = k_{m-1}$ for $j >= m$. Then x is composed of the m bits of k right justified with n-m MSB repetitions left justified to make up an n bit word. The interpretation of x is as two's complement integer.

output = x

latency = 1 bit

Restrictions:

1. For correct function parameter 1 must be the same as the system word-length.

2. ctrl must be c1 control.

Simulator Checks and Warnings

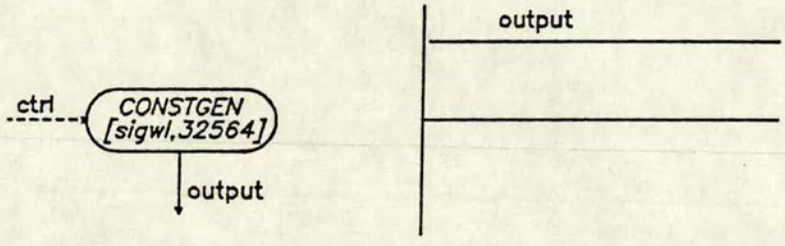
Checks are made for input synchronisation timing.

1.3.3. Constgen

Description:

Each c1 cycle, consgen outputs, LSB first, the m bits defined by the user chosen value of parameter 2 followed by (n-m) MSB repetitions to make up the full n bit constant word. Constgen is, in effect, a one word bit-serial ROM. The use of constgen in fixed coefficient designs saves generating required constant values off chip.

Flow Diagram & Simulation Pattern:



Syntax:

Constgen [sigwl,constspec] (ctrl) -> output

Parameters:

Let n=swl, m=sigwl, k=constspec

param	min	max	constraint	meaning
1	1	32	integer	significant word-length
2	*	*	integer	coded constant

Circuit Design

The adder circuit design was adapted from the programmable add-subtract circuit used in the multiplier. It uses data select pass transistor EXOR networks, giving minimum transistor count and power dissipation. This style of design can, however, give rise to speed and testability problems.

Restrictions:

1. For addition to function correctly the LSB of each addend and of the carryin must have the same significance. The LSB of the sum will have this significance also.

2. If each addend has one guard bit (i.e., the two MSBs are identical) then no overflow or underflow can occur.

3. Two word addition causes arithmetic growth of one bit. Overflow in the previous sum is flagged by the single bit value of caryout at LSB time, with value 0 signifying no overflow/underflow and value 1 signifying overflow/underflow.

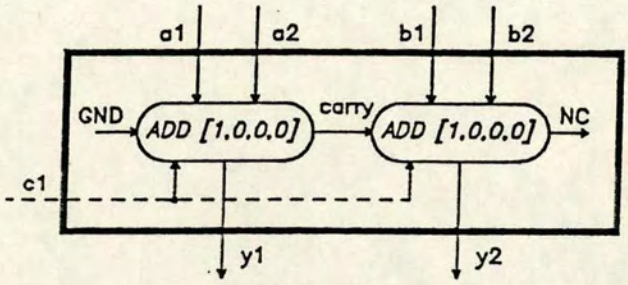
4. For single precision addition the carryin input is connected to ground.

5. The carryin/carryout configuration is designed for easy construction of multiple precision addition

6. For correct function the ctrlin input must be LSB or c1 control.

Simulator Checks and Warnings

Checks are made for overflow and underflow and for input synchronisation timing.



F is the word of bitwise addition carry bits generated from the addition, and has a fixed latency of only one bit. More formally, if

$$A = a_0.2^0 + a_1.2^1 + \ldots + a_{n-2}.2^{n-2} - a_{n-1}.2^{n-1}$$
$$B = b_0.2^0 + b_1.2^1 + \ldots + b_{n-2}.2^{n-2} - b_{n-1}.2^{n-1}$$
$$C = c_0.2^0 + c_1.2^1 + \ldots + c_{n-2}.2^{n-2} - c_{n-1}.2^{n-1}$$

then F is given by

$$F = f_0.2^0 + f_1.2^1 + \ldots + f_{n-2}.2^{n-2} - f_{n-1}.2^{n-1}$$

where :

$$f_1 = c_0 \text{ AND } (a_0 \text{ XOR } b_0) \text{ OR } (a_0 \text{ AND } b_0)$$
$$f_j = f_{j-1} \text{ AND } (a_{j-1} \text{ XOR } b_{j-1}) \text{ OR } (a_{j-1} \text{ AND } b_{j-1})$$

where $1<=j<=n-1$ and

$$f_0 = f_{n-1} \text{ AND } (a_{n-1} \text{ XOR } b_{n-1}) \text{ OR } (a_{n-1} \text{ AND } b_{n-1})$$

Note that, in this final expression for f_0 the values a_{n-1} and b_{n-1} refer to the previous input words and f_{n-1} the corresponding previous word carry bit. Thus f_0 is an overflow/underflow flag for the previous addition.

param	min	max	constraint	meaning
1	1	32	integer	latency of sum
2	0	1	integer	predelay on addend1
3	0	1	integer	predelay on addend2
4	0	1	integer	predelay on carryin

Function:

sum = addend1 + addend2 + (carryin & 1)

latency = user chosen value of parameter 1

carryout : see below

latency = 1 bit

For notational convenience, suppose we have:

add[m,0,0,0](cLSB) A,B,C -> S,F

and let swl=n. Then the addends A, B are assumed to have the same interpretation i.e., LSBs have the same significance. S is the sum of A and B with only the LSB of C added in. In normal use C will be replaced by GND so that this LSB is zero. Adders can, however, be cascaded to form multiple precision accumulation. This is achieved by connecting the carryout from the previous adder to the carryin of the next as shown in the figure below, which illustrates adders configured for double precision addition. In this case the C input is used to propagate any carry generated from the addition of less significant bytes into the addition of the more significant bytes. The first adder has its C input replaced by GND.

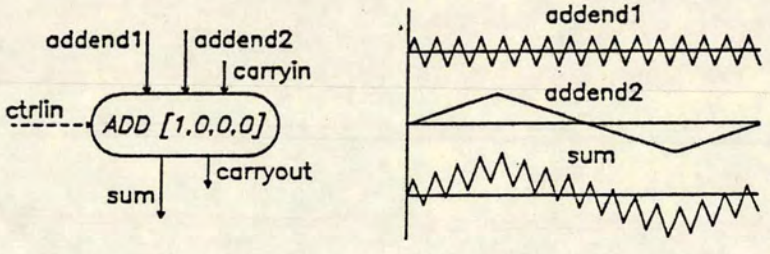
The latency of the sum can be user chosen by means of the first parameter.

1.3.2. Add

Description

Add forms the bit-serial sum of the first two inputs. The third input is an external carry, which is strobed at LSB time to initialise the adder. This input is normally connected to GND, but can be used to construct multiple precision adders (see function). The second output, the word of carry bits, is provided for the same purpose. The latency of the sum output can have any user chosen value in the allowed range.

Flow Diagram & Simulation Pattern:



Syntax:

```
add [latency,del1,del2,del3] (ctrlin) -
        addend1,addend2,carryin -> sum,carryout
```

Parameters:

complement) rather than inversion and addition of one bit at LSB significance, (two's complement) was made in order to achieve a hardware saving of a half adder. This saving is not significant but the added accuracy was not needed for the applications for which the primitive was originally designed. In a comprehensive cell library another variant of absolute might be made available, giving the accurate two's complement version.

Function:

```
output = input          if MSB=0
       = NOT(input)     if MSB=1

latency =  (swl + 3) bits
```

Restrictions:

1. The value of parameter 1 must be the same as the system word-length for correct function.

2. If the sign bit of the input is negative (MSB=1) then the operation of absolute is to invert the input. This one's complement function gives rise to a one bit error at LSB significance in the output, but results in smaller hardware area (see section on circuit design). In principle there is no reason why a two's complement version could not also be available.

3. For correct operation ctrl must be c1.

Simulator Checks and Warnings

Checks are made for illegal values of parameter 1 and for input synchronisation timing.

Circuit Design

The circuit is implemented as a shift register to store the input word, a sign detect and programmable output inversion circuit followed by an output drive stage. The choice of inversion (one's

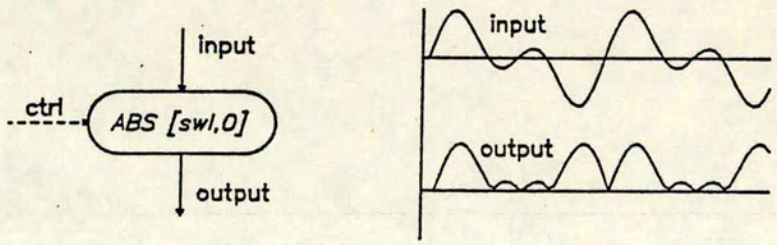
1.3. ARITHMETIC PRIMITIVES

1.3.1. Absolute

Description:

The absolute primitive performs the modulus or full-wave rectify function on two's complement data. It outputs the input signal if this is non-negative or the inverted signal if it is negative.

Flow Diagram & Simulation Pattern:



Syntax:

absolute [swl,del] (ctrl) input -> output

Parameters:

param	min	max	constraint	meaning
1	6	32	integer	system word-length
2	0	1	integer	predelay on input

where w is a number of words and b is a number of bits with 0<=b<n then Y(T) is defined when (T mod n)=b.

Usually the values are just interpreted as words X and Y, with latency d. However, if reference to a specific bit at a specific input time is needed then this can easily be done, using the above notation. For simplicity, values are interpreted throughout as being integer. Other interpretations are explained where appropriate. Note that the abbreviation swl, for system word-length, is used frequently.

functioning. In order to simplify the notation, the node and the value on it are given the same name. As a value this is interpreted as an arbitrary word value at an arbitrary time. The values of the inputs will all occur simultaneously (i.e., each primitive has time-aligned inputs). Each resulting output value is defined with an associated latency, which is the delay from the input time to the time when the corresponding output occurs (the outputs of a primitive may not be time aligned). This notation can be directly derived from a more rigorous notation specifying distinct node names, bit and word values at specific times. The latter notation is required when more complicated reference to bit values or internal state values is needed in the definition of function. This latter notation is as follows.

Let T=0,1,2,... denote bit time. Suppose that this time is local to the inputs of the primitive in question, so that at time T=0 the LSB appears at the inputs. Let the system word-length be n and suppose that X is an input node. Further let T=t where (t mod n)=0 then the value on X at time t is:

$$X = x(t).2^0 + x(t+1).2^1 + \ldots + x(t+n-2).2^{n-2} - x(t+n-1).2^{n-1}$$

where x(j) = 0,1. This is often abbreviated to:

$$X = x_0.2^0 + x_1.2^1 + \ldots + x_{n-2}.2^{n-2} - x_{n-1}.2^{n-1}$$

These expressions represent the 2's complement integer interpretation of the bit stream which occurs at X from T=t till T=t+n-1.

If the latency of an output node Y relative to X is d=nw+b

information would enable a skilled circuit designer to enter new primitives into the compiler, given the system requirement for such. The contents of this section are not of primary concern to the system designer who wishes only to use the compiler.

The primitives fall into the following categories: arithmetic function, storage, control, formatting and miscellaneous. Within these functional groupings they are presented alphabetically for convenience of reference. For each primitive a brief description is given. A flow diagram representation is shown with a simulation pattern. The FIRST HDL syntax is then listed. A short explanation of the parameters, a definition of function and notes on restrictions for valid use are given. The checks that the simulator carries out are listed. These greatly aid system description debugging and are not, in general, duplicated by the corresponding hardware. In addition to valid node connectedness checks the most frequently made check is for input timing synchronisation, which identifies latency inconsistencies in a design. Finally some relevant notes on circuit design are added.

1.2. Notation

Some notation is required for defining primitive function. This can be more or less formal. In particular, each primitive input or output node will have a name and at each of these named nodes bit streams, grouped into words as marked by the associated LSB or c1 control, will appear throughout the time the primitive is

List of Primitives

Arithmetic	Storage	Control	Format
Absolute	Bitdelay	Cbitdelay	Fformat1to1
Add	Worddelay	Controlgenrator	Fformat2to1
Constgen		CWorddelay	Fformat3to1
Dpmultiply			format1to2
Dshift			
Mshift			flimit
Multiplex			
Multiply			
Order			
Subtract			

Miscellaneous

Padin
Padout
Clock generator
Substrate bias generator

Table III.1

for its use. As this text is an introduction to the design methodology underlying such a compiler and to the system design techniques that can be applied with it, the technical circuit and programming issues are not dealt with in detail.

In the first sub-section the notation is explained which is used to describe the function and operation of the primitives. Next, in the main numbered sections of the chapter, a specific primitive library is defined and described. Finally, the chapter ends with a short section which indicates how primitives are designed and defines their formal interfaces to the compiler. This section is intended to give a flavour of what is involved in building the primitive library as part of a silicon compiler. Such

1. PRIMITIVE LIBRARY

1.1. Introduction

In this chapter the elements of a bit-serial primitive library are described. Primitives are the lowest functional elements in the system hierarchy. They exist in hardware and all systems will be constructed from them. In the abstract case it may be possible to define a canonic or covering set of such primitives, which will deal with all problems. However, practical requirements for hardware and computational efficiency often demand that special solutions be developed for particular applications. The approach adopted has been to build a library of primitives to cover the range of applications of most interest to the specific applications of signal processing. Where no primitive exists to meet a requirement then it is necessary to custom design and verify one and then add it to the library. Our experiments in system design using the techniques outlined in this text demonstrate that a wide range of applications can be satisfied using a set of a dozen or so primitives. These are listed in Table III.1.

It is a primary feature of this library that all of the composed primitives obey the geometric, electrical and signaling conventions set out in chapter 3. The details of the design of each primitive are technology specific and each primitive library requires a separate technical accompanying document. This are needed for documentation and maintenance of the compiler, but not

APPENDIX II : THE FIRST PRIMITIVE LIBRARY

ABSTRACT

This appendix describes the FIRST primitive hardware. The primitive hardware was specified by P. B. Denyer and D. Renshaw and was designed by D. Renshaw, H. S. C. Wallace, D. J. Talbot and N. Henderson. Hardware was tested by D. Renshaw and N. Henderson. The contents of this appendix was written by D. Renshaw; it appears in a reduced form in "VLSI Signal Processing: A Bit-Serial Approach" by P. B. Denyer and D. Renshaw (Addison-Wesley).

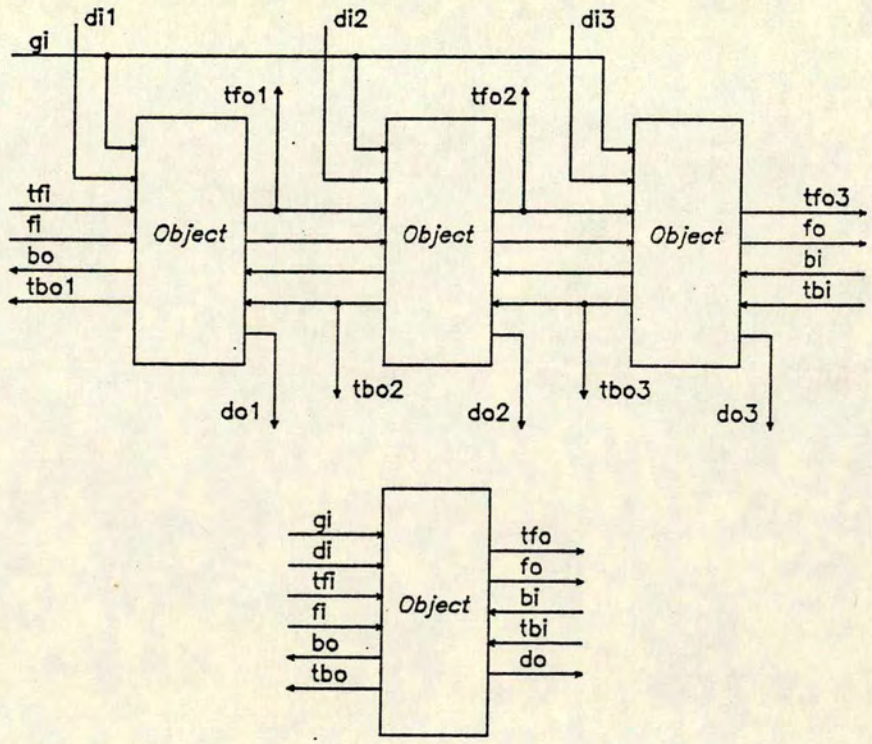


Figure 4

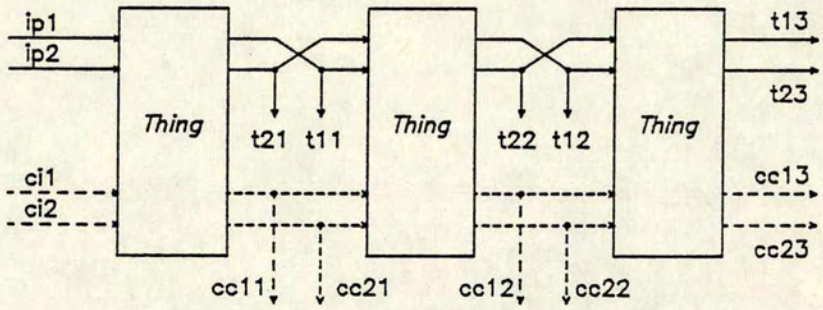


Figure 5

```
!================================================================
OPERATOR F7  di1,di2,di3,gi,fi,bi,tfi,tbi -> -
     do1,do2,do3,fo,bo,tfi1,tfi2,tfi3,tbo1,tbo2,tbo3

     SIGNAL di, do, tfo, tbo

     Object [1] di,gi,fi,bi,tfi,tbi -> do,fo,bo,tfo,tbo -
     TIMES 3 WITH -
       fo -> fi -
       tfo => tfi = tfi1, tfi2, tfi3 -
       bi <- bo -
       tbi <= tbo = tbo1, tbo2, tbo3
     di = di1, di2, di3
     do = do1, do2, do3
END

OPERATOR F8 (ci1, ci2 -> cc11 THROUGH 13, cc21 THROUGH 23) -
     i1, i2 -> t11 THROUGH 13, t21 THROUGH 23

     SIGNAL o1, o2
     CONTROL co1, co2

     Thing [1] (ci1,ci2->co1,co2) i1,i2 -> o1,o2 TIMES 3 WITH -
       (ci1, ci2 => co1 = cc11 THROUGH 13 : co2 = cc21 THROUGH 23) -
       o1, o2 => i2 = t11 THROUGH 13 : i2 = t21 THROUGH 23

END
!================================================================
```

Note that in the first of these examples the signal names have been chosen with a mnemonic letter code as follows.

gi	global input
di	distinct inputs
do	distinct outputs
fi	forward cascade input
fo	forward cascade output
bi	backward cascade input
bo	backward cascade output
tfi	tapped forward cascade input
tfo	tapped forward cascade outputs
tbi	tapped backward cascade input
tbo	tapped backward cascade outputs

In the second example there is more than one signal connected in the same cascade mode. All such signals must be named in the same phrase, as lists of nodes followed by corresponding lists of tap names. The lists of tap names are separated by the : symbol.

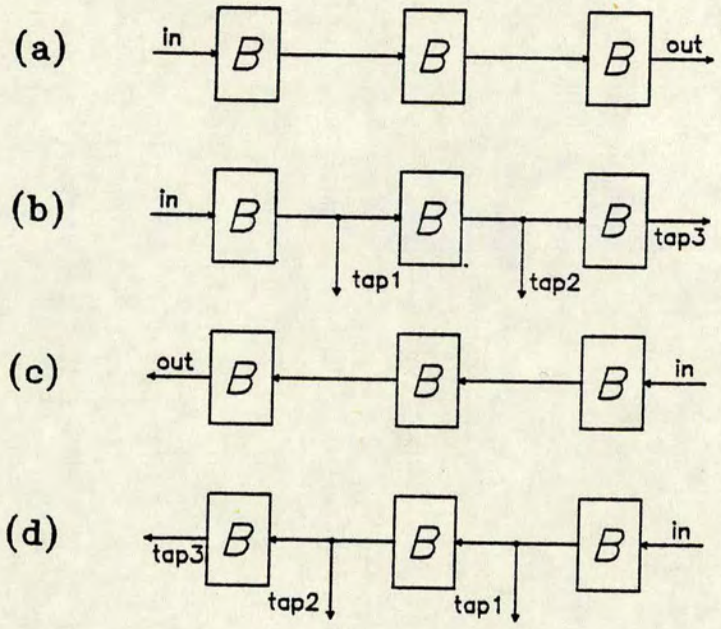


Figure 3

nodes must be named for each of the tapped cascade configurations. This occurs through the phrase starting with the = symbol and followed by the list of nodes which are to be used for each of the tap points.

Figures 4 and 5 show two further examples which combine several different types of connection in one structure and two cascades of connection of the same type, respectively. The associated syntax follows.